Automatische Generierung von effizienten, parallelen Implementierungsvarianten für ODE-Verfahren aus deren Datenflussgraphen mit Kernelfusion und Tiling für GPUs und CPUs

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von
Tim Werner
aus Pegnitz

1. Gutachter: Prof. Dr. Matthias Korch
2. Gutachter: Prof. Dr. Michael Guthe
3. Gutachter: Prof. Dr. Martin Bücker

Tag der Einreichung: 11.1.2023
Tag des Kolloquiums: 17.5.2023

Danksagung

Ich möchte mich zunächst bei allen, die zum Gelingen dieser Arbeit beigetragen haben bedanken. Insbesondere gilt mein Dank folgenden Personen:

- Herrn Prof. Dr. Matthias Korch mir die Möglichkeit zur Promotion zu geben, für die sehr ausgiebige und hochwertige wissenschaftliche Betreuung während dieser Promotion und für die sehr liebevolle emotionale Unterstützung nach dem Tod meiner beiden Eltern.
- Herrn Simon Melzner für die sehr gute technische Betreuung während dieser Promotion und ebenso für die sehr liebevolle emotionale Unterstützung nach dem Tod meiner beiden Eltern.
- Herrn Prof. Dr. Thomas Rauber mir die Ressourcen seines Lehrstuhls zur Verfügung zu stellen und für das Feedback zu dieser Arbeit.
- Allen meinen übrigen Kollegen an dem Lehrstuhl Angewandte Informatik II für das sehr angenehme und freundschaftliche Arbeitsklima.

Zusammenfassung

Das Lösen von Anfangswertproblemen mit Hilfe von ODE-Verfahren ist in vielen wissenschaftlichen Bereichen und für ingenieurtechnische Anwendungen zwingend erforderlich. Bedauerlicherweise gibt es eine große Anzahl von relevanten Anfangswertproblemen, ODE-Verfahren und Prozessorarchitekturen, wobei potentiell jede Kombination von Anfangswertproblem, ODE-Verfahren und Prozessorarchitektur von einer speziell dafür optimierten Implementierungsvariante profitiert. Wegen dieser großen Anzahl an Kombinationen ist es sehr zeitaufwändig, per Hand eine optimierte Implementierungsvariante für jede davon zu erstellen. Eine solche optimierte Implementierungsvariante ist jedoch erforderlich, sofern man die Rechenleistung eines Prozessors gut ausnutzen möchte.

In dieser Arbeit befassen wir uns damit, wie wir automatisch optimierte Implementierungsvarianten von ODE-Verfahren für das Lösen von Anfangswertproblemen generieren können. Dabei ist es unser Ziel ein breites Spektrum an ODE-Verfahren, allgemeine Anfangswertprobleme sowie solche mit einer ausnutzbaren Zugriffsstruktur und GPUs sowie CPUs als optimierte Zielplattformen zu unterstützen. Hierfür zeigen wir zunächst, dass sich viele häufig verwendete ODE-Verfahren, wie die RK-Verfahren, die PIRK-Verfahren oder die Peer-Verfahren, zu einem Datenflussgraphen bestehend aus Auswertungen der problemspezifischen rechten Seite (RHS), Linearkombinationen (LC) und Reduktionen (RED) als Grundoperationen abstrahieren lassen. Bei vielen Verfahren besteht der Datenflussgraph zusätzlich aus einer Abhängigkeitskette aus RHS $\rightarrow$ LC-Gliedern, in welcher voneinander unabhängige Grundoperationen zum selben RHS $\rightarrow$ LC-Glied zusammengefasst werden.

Diesen Datenflussgraphen können wir nun leicht zur Generierung einer Basisimplementierung für das Lösen eines Anfangswertproblems verwenden, welche eine Grundoperation nach der anderen über je ein paralleles Kernel berechnet. Ein solches Kernel nutzt die Systemparallelität innerhalb einer Grundoperation aus, um deren Berechnungen auf sämtlichen Rechenkernen der Zielplattform auszuführen. Für diese Basisimplementierung stellen wir von uns durchgeführte experimentelle Untersuchungen auf einer Test-GPU und einer Test-CPU vor. Diese experimentellen Untersuchungen zeigen, dass eine solche Basisimplementierung für dichtbesetzte Anfangswertprobleme mit einer hohen arithmetischen Intensität, wie das N-Körperproblem, bereits eine gute Performance erzielen kann. Dahingegen ist sie für dünnbesetzte Anfangswertprobleme mit einer geringen arithmetischen Intensität, wie unser Testproblem BRUSS2D oder viele andere Stencil-Probleme, durch die DRAM-Bandbreite gebunden und kann die Rechenwerke der Test-GPU und Test-CPU nur schlecht auslasten.

Deswegen verwenden wir als Nächstes die Technik der Kernelfusion, um für allgemeine, dünnbesetzte Anfangswertprobleme mit einer geringen arithmetischen Intensität die Datenwiederverwendung und damit die Performance zu erhöhen. Mit Hilfe dieser Technik der Kernelfusion verschmelzen wir jeweils mehrere Grundoperationen im Datenflussgraphen zu einem Kernel, um innerhalb eines solchen fusionierten Kernels die geladenen Daten über den On-Chip-Speicher wiederzuverwenden, die erzeugten Daten über den On-Chip-Speicher zu transferieren und ein unnötiges Zurückschreiben von den erzeugten Daten in den DRAM zu vermeiden. Dabei können wir jeweils sämtliche Grundoperationen in einem RHS $\rightarrow$ LC-Glied zu einem Kernel verschmelzen. Jedoch darf eine RHS-Operation bei einem allgemeinen Anfangswertproblem für die Auswertung einer ihrer Komponenten beliebig auf ihren Argumentvektor zugreifen. Deshalb können wir keine LC $\rightarrow$ RHS-Abhängigkeiten und Grundoperationen aus unterschiedlichen RHS $\rightarrow$ LC-Gliedern miteinander verschmelzen. Zusätzlich wenden wir vor der Fusion zwei Enabling-Transformationen an, und zwar die Splitting-Transformation zum

Aufspalten von LC-Operationen und die Cloning-Transformation zur redundanten Berechnung einer Grundoperation. Als Nächstes stellen wir vor, wie sich eine Variante mit Kernelfusion sowohl auf GPUs als auch auf CPUs mit architekturspezifischen Optimierungen automatisch generieren lässt. Schließlich führen wir eine experimentelle Untersuchung der per Kernelfusion und per Enabling-Transformationen generierten Implementierungsvarianten von mehreren expliziten ODE-Verfahren mit dem Testproblem BRUSS2D auf mehreren modernen GPUs und CPUs durch. Diese Untersuchungen zeigen, dass sich auf den getesteten GPUs durch die Kernelfusion und Enabling-Transformationen eine Beschleunigung zwischen 1.62 und 3.32 gegenüber der jeweiligen Basisimplementierung des ODE-Verfahrens erzielen lässt, während sich auf den getesteten CPUs eine Beschleunigung zwischen 1.98 und 4.14 gegenüber der jeweiligen Basisimplementierung des ODE-Verfahrens erzielen lässt. Diese Beschleunigung ist darauf zurückzuführen, dass die Kernelfusion und die Enabling-Transformationen zusammen die zwischen Prozessor und DRAM transferierte Datenmenge je nach ODE-Verfahren auf 50 % bis 25 % gegenüber der jeweiligen Basisimplementierung reduzieren können. Allerdings zeigen die Experimente auch, dass die per Kernelfusion und Enabling-Transformationen generierten Implementierungsvarianten auf den getesteten modernen GPUs und CPUs weiterhin durch die DRAM-Bandbreite gebunden sind.

Da das beliebige Zugriffsmuster der RHS-Operationen für ein allgemeines Anfangswertproblem weitere Lokalitätsoptimierungen verhindert, spezialisieren wir uns, um solche weiteren Lokalitätsoptimierungen durchzuführen, als Nächstes auf dünnbesetzte rechte Seiten mit einer beschränkten Zugriffsdistanz. Bei diesen darf eine RHS-Operation für die Auswertung einer gegebenen Komponente nur auf diejenigen Komponenten des Argumentvektors, die sich innerhalb der beschränkten Zugriffsdistanz um der gegebenen Komponente herum befinden, zugreifen. Dies erlaubt es uns LC $\rightarrow$ RHS-Abhängigkeiten über eine partitionierte Fusion zu verschmelzen. Diese partitionierte Fusion können wir nun entlang der LC $\rightarrow$ RHS-Abhängigkeitskette des ODE-Verfahrens anwenden, wodurch sich zwei Tiling-Schemata ergeben, und zwar das trapezoidale Tiling-Schema und das hexagonale Tiling-Schema. Innerhalb eines Kernels mit Tiling kann ein Prozessor nun wieder analog, wie bei einem per Kernelfusion generierten Kernel, Daten wiederverwenden. Wir ermöglichen dieses Tiling nicht nur über die Stufen eines ODE-Verfahrens, sondern auch über dessen Zeitschritte. Ebenso stellen wir vor, wie mehrere Rechenkerne eines Prozessors an einem Tile zusammenarbeiten können, um darüber die maximale Tile-Größe und die Datenwiederverwendung zu erhöhen. Zusätzlich entwickeln wir einen Autotuning-Ansatz, um für beide Tiling-Schemata die optimalen Tile-Größen zu bestimmen. Als Nächstes gehen wir darauf ein, wie sich eine Variante mit Tiling sowohl auf GPUs als auch auf CPUs mit architekturspezifischen Optimierungen implementieren lässt. Schließlich führen wir für mehrere explizite ODE-Verfahren eine experimentelle Untersuchung der Implementierungsvarianten mit Tiling und mit BRUSS2D als Testproblem auf mehreren modernen GPUs und CPUs durch. Diese Untersuchung zeigt, dass sich durch das Tiling sowohl die Datenwiederverwendung als auch die Performance sowohl auf den getesteten GPUs als auch auf den getesteten CPUs gegenüber den Implementierungsvarianten mit Kernelfusion stark erhöhen lassen. So erzielen für die getesteten ODE-Verfahren die Implementierungsvarianten mit Tiling gegenüber einer durch Kernelfusion generierten Referenzimplementierung bei einer kleinen Zugriffsdistanz von 32 auf den getesteten GPUs eine Beschleunigung von 1.8 bis 8.2 und auf den getesteten CPUs eine Beschleunigung von 7.5 bis 14.1. Ebenso zeigen die Untersuchungen, dass es bereits bei kleinen bis mittelgroßen Zugriffsdistanzen für die Performance einer Variante mit Tiling sehr vorteilhaft ist, wenn ein Tile nicht nur von einem einzelnen Kern abgearbeitet wird, sondern wenn jeweils mehrere Rechenkerne an einem Tile zusammen-

arbeiten. Durch dieses Zusammenarbeiten von mehreren Rechenkernen an einem Tile lässt sich zum Beispiel auf der getesteten Volta-GPU und Skylake-CPU bei einer beschränkten Zugriffsdistanz von 131 072 noch eine Beschleunigung von 1.4 beziehungsweise 1.6 gegenüber der per Kernelfusion generierten Referenzimplementierung erzielen.

Wir stellen zudem ein von uns entwickeltes automatisiertes Framework vor, welches sowohl die Generierung als auch die Ausführung der Varianten der Basisimplementierung, der Varianten mit Kernelfusion sowie der Varianten mit Tiling anhand des Datenflussgraphen eines ODE-Verfahrens durchführt. Dieses Framework unterstützt sowohl GPUs als auch CPUs als optimierte Zielplattformen und implementiert zusätzlich eine Reihe von hilfreichen Funktionalitäten, wie ein automatisches Speichermanagement, eine NUMA-Unterstützung für CPUs und eine Autotuning-Funktionalität für das Tiling. Dieses Framework beinhaltet zudem eine DSL, mit welcher ein Benutzer leicht ODE-Verfahren zum Framework hinzufügen kann.

Abstract

Solving initial value problems using ODE methods is essential in many scientific fields and for engineering applications. Unfortunately, there are a large number of relevant initial value problems, ODE methods and processor architectures, with potentially every combination of initial value problem, ODE method and processor architecture benefiting from a specially optimised implementation variant. Because of this large number of combinations, it is very time-consuming to manually create an optimised implementation variant for each of them. However, such an optimised implementation variant is necessary if one wants to make good use of the computing power of the processor.

In this thesis, we address how we can automatically generate optimised implementation variants of ODE methods for solving initial value problems. Our goal is to support a wide range of ODE methods, general initial value problems with an arbitrary memory access pattern, initial value problems with limited access distance, and GPUs as well as CPUs as optimised target platforms. To this end, we first show that many commonly used ODE methods, such as RK methods, PIRK-Methods or Peer-Methods, can be abstracted by a dataflow graph consisting of evaluations of the problem-specific right-hand side (RHS), linear combinations (LC) and reductions (RED) as basic operations. For many methods, the data flow graph additionally consists of a dependency chain of RHS $\rightarrow$ LC-links, in which basic operations being independent of each other are inserted into the same RHS $\rightarrow$ LC-link.

We can now easily use this data flow graph to generate a basic implementation for solving an initial value problem, which computes one basic operation at a time via one parallel kernel each. Such a parallel kernel exploits the system parallelism within a basic operation to perform its computations on all computational cores of the target platform. For this basic implementation, we conduct an experimental evaluation on a test GPU and test CPU. This experimental evaluation shows that such a basic implementation can already achieve a good performance for densely populated initial value problems with a high arithmetic intensity, such as the N-body problem. On the other hand, for sparse initial value problems with a low arithmetic intensity, such as our test problem BRUSS2D or many other stencil problems, it is bound by the DRAM bandwidth and can only poorly utilise the execution units of the processor.

Therefore, we next use the technique of kernel fusion to increase the data reuse for general sparse initial value problems with a low arithmetic intensity. Using this technique, we can fuse groups of basic operations in the data flow graph into one kernel each. Such a fused kernel reuses its loaded data via the on-chip memory of the processor, transfers its generated data via the on-chip memory of the processor and avoids unnecessary write-backs of its generated data to DRAM. In doing so, we can fuse all the basic operation of a RHS $\rightarrow$ LC-link into one kernel each. However, for general initial value problems, an RHS operation may arbitrarily access its argument vector for evaluating one of its components. Therefore, we cannot fuse LC $\rightarrow$ RHS-dependencies and basic operations from different RHS $\rightarrow$ LC-links. In addition, we apply two enabling transformations before the fusion, namely the splitting transformation to split LC operations and the cloning transformation to redundantly compute a basic operation. Next, we present how a variant can be automatically generated with kernel fusion on both GPUs and CPUs with architecture-specific optimisations. Finally, for several explicit ODE methods, we perform an experimental evaluation of the implementation variants generated via kernel fusion and enabling transformations with the test problem BRUSS2D on several modern GPUs and CPUs. This evaluation shows that on GPUs, depending on the method and the GPU used, the kernel fusion and enabling transformations can achieve a speedup between 1.62 and 3.32

compared to the respective basic implementation of the method, while on CPUs, depending on the method and the CPU used, a speedup between 1.98 and 4.14 can be achieved compared to the respective basic implementation of the method. This speedup is due to the fact that, depending on the method, the kernel fusion and enabling transformations can reduce the amount of data transferred between the processor and DRAM to between 50 % and 25 % compared to the respective basic implementation of the method. However, the experiments also show that the implementation variants generated by kernel fusion and enabling transformations are still bound by the DRAM bandwidth on the tested modern GPUs and CPUs.

Since the arbitrary access pattern of RHS operations for general initial value problems prevents further locality optimisations, we next specialise in RHS with a limited access distance. Here, an RHS evaluation of a single component may only access those components of the argument vector that are within the limited access distance around this given component. This allows us to fuse LC $\rightarrow$ RHS-dependencies via a partitioned fusion. We can now apply this partitioned fusion along the dependency chain of the ODE method, resulting in two tiling schemes, namely the trapezoidal tiling scheme and the hexagonal tiling scheme. In a kernel generated via tiling, a processor can now reuse data again analogously to a kernel generated via kernel fusion. We realize this tiling not only across the stages of an ODE method, but also across its time steps. We also introduce how multiple cores of a processor can cooperate on a tile to increase the maximum tile size and thereby the data reuse. Next, we address how a variant with tiling can be implemented on both GPUs and CPUs with architecture-specific optimisations. Additionally, we develop an autotuning approach, which determines the optimal tile sizes for both tiling schemes. Finally, for several explicit ODE methods, we conduct an experimental evaluation of the implementation variants generated via tiling with BRUSS2D as a test problem on several modern GPUs and CPUs. This evaluation shows that the variants generated via tiling can greatly increase both performance and data reuse on both the test GPUs and test CPUs compared to the variants generated via kernel fusion. For example, for the tested ODE methods, for a small limited access distance of 32 and on the tested GPUs the variants with tiling achieve a speedup of 1.8 to 8.2 compared to a reference implementation generated by kernel fusion. Also on the tested CPUs for the same small limited access distance of 32 the variants with tiling achieve a speedup of 7.5 to 14.1 compared to a reference implementation generated by kernel fusion. The experimental evaluation also shows that even for small to medium access distances, it is very advantageous for the performance of the variants with tiling if a tile is not only processed by a single core, but if several cores cooperate on one tile. By having several cores of the Volta GPU or of the Skylake CPU cooperate on the same tile, for a large limited access distance of 131 072 a speedup of 1.4 and 1.6 respectively can be achieved compared to the reference implementation generated by kernel fusion.

For this thesis, we implemented the generation and the execution of the variants of the basic implementation, the variants with fusion and the variants with tiling via the data flow graph abstraction of an ODE method by an automatic framework. This framework supports both GPUs and CPUs as target platforms and additionally implements several of helpful functionalities, such as automatic memory management, NUMA support for CPUs and an autotuning functionality for the tiling. Additionally, this framework includes an DSL, which allows a user to easily add new ODE methods to the framework.

INHALTSVERZEICHNIS

1 Einleitung **17**
1.1 Motivation . . . 17
1.2 Zielstellung . . . 19
1.3 Vorgehen . . . 19
1.3.1 Abstraktion von ODE-Verfahren durch einen Datenflussgraphen . . . 19
1.3.2 Erzeugung der Basisimplementierungen aus dem Datenflussgraphen eines ODE-Verfahrens . . . 19
1.3.3 Erhöhung der Datenwiederverwendung durch die Kernelfusion für allgemeine Anfangswertprobleme . . . 20
1.3.4 Erhöhung der Zugriffslokalität durch Tiling für Anfangswertprobleme mit beschränkter Zugriffsdistanz . . . 21
1.3.5 Implementierung unseres Ansatzes in einem Framework . . . 22
1.4 Gliederung . . . 23
1.5 Zugehörige Publikationen . . . 25

2 Verwandte Arbeiten **26**
2.1 Arbeiten über die Entwicklung neuer ODE-Verfahren . . . 26
2.2 Arbeiten im Themenbereich des Autotunings . . . 26
2.3 Arbeiten im Themenbereich der Kernelfusion . . . 29
2.4 Arbeiten im Themenbereich der Stencil-Codes und des Tilings . . . 30
2.5 Arbeiten im Themenbereich der optimierten Implementierungen für ODE-Verfahren . . . 32
2.6 Einordnung unserer folgenden Arbeit . . . 34

3 Moderne GPU-Architekturen **36**
3.1 Allgemeines . . . 36
3.2 Terminologie . . . 37
3.3 SIMD . . . 38
3.4 Aufbau . . . 39
3.5 Kernelausführung . . . 45
3.6 Kontrollfluss . . . 46
3.7 Speicherräume . . . 50
3.8 Speicherzugriffe . . . 52
3.9 Latenzüberbrückung . . . 54
3.10 Multithreading mit einer variablen Anzahl von parallelen Threads . . . 55
3.11 Occupancy . . . 56
3.12 Task-Queues und Events . . . 57
3.13 Programmierung von GPUs über GPU-APIs . . . 57
3.14 Vergleich zwischen modernen GPU-Architekturen und modernen x64-CPU-Architekturen . . . 58
3.15 Fazit . . . 60

4 Grundlagen zu ODE-Verfahren **61**
4.1 Allgemeines . . . 61
4.2 Klassifikation von ODE-Verfahren . . . 61

4.3 Parallelität in ODE-Verfahren 62
4.4 Lokaler Diskretisierungsfehler und Ordnung 63
4.5 Schrittweitensteuerung 64
4.6 Explizite RK-Verfahren 65
4.7 Implizite RK-Verfahren 67
4.8 PIRK-Verfahren 68
4.9 Peer-Verfahren 69
4.10 Adams-Bashforth-Verfahren 70
4.11 Richardson-Extrapolationsverfahren 71
4.12 Allgemeine lineare Verfahren 71
4.12.1 Grundlegendes 71
4.12.2 RK-Verfahren als allgemeine lineare Verfahren 72
4.12.3 Adams-Bashforth-Verfahren als allgemeine lineare Verfahren 73

5 Klassen von Anfangswertproblemen und Auswahl repräsentativer Testprobleme 74
5.1 Klassifikation von Anfangswertproblemen 74
5.2 Testproblem BRUSS2D 81
5.3 Testproblem NBODY 83

6 Datenflussgraphrepräsentation für ODE-Verfahren 85
6.1 Aufbau eines Datenflussgraphen 85
6.2 Abhängigkeiten zwischen Grundoperationen 87
6.3 Beschreibung von Abhängigkeiten über die Zeitschrittgrenzen 88
6.4 Datenflussgraphen der Verfahrensklassen 89
6.5 Einfache mathematische Optimierungen auf dem Datenflussgraphen 90
6.6 Abhängigkeitskette als eine Struktur im Datenflussgraphen 91

7 Grundkonzepte unserer Implementierungen 96
7.1 Motivation 96
7.2 Abstrakte Kernels 96
7.3 Permanente Vektoren 96
7.4 Ausführbare Objekte 97

8 Basisimplementierung 98
8.1 Motivation 98
8.2 Theorie 98
8.3 Basisimplementierung auf GPUs 99
8.3.1 Implementierung 99
8.3.2 Experimente 102
8.3.2.1 Aufbau und Durchführung der Experimente 102
8.3.2.2 Diskussion der Messergebnisse 102
8.4 Basisimplementierung auf CPUs 104
8.4.1 Implementierung 104
8.4.2 Experimente 107
8.4.2.1 Aufbau und Durchführung der Experimente 107
8.4.2.2 Diskussion der Messergebnisse 107
8.5 Fazit 110

9 Kernelfusion 111
9.1 Motivation 111
9.2 Theorie 111
9.2.1 Überblick 111
9.2.2 Annahmen zur Kernelfusion für ein Performance-Modell 112
9.2.3 Regeln für die Kernelfusion 114
9.2.4 Speicheroptimierungen in einem Kernel 117
9.2.5 Mögliche Fusionen in einer Abhängigkeitskette 119
9.2.6 Enabling-Transformationen 120
9.2.7 Anwendung der Splitting-Transformation auf eine Abhängigkeitskette . . 124
9.2.8 Anwendung der Kernelfusion auf mehrere ODE-Verfahren 125
9.2.8.1 Verner-Verfahren 126
9.2.8.2 Verfahren von Dormand und Prince 129
9.2.8.3 PIRK-Verfahren 132
9.2.8.4 Peer-Verfahren 135
9.3 Kernelfusion auf GPUs 138
9.3.1 Implementierung 138
9.3.1.1 Grundlegendes 138
9.3.1.2 Ausnutzung des On-Chip-Speichers 138
9.3.1.3 Weitere Überlegungen zum Mapping zwischen Vektorkomponenten und Workitems und zu der Dimensionalität eines fusionierten Kernels 139
9.3.1.4 Codegenerator für fusionierte GPU-Kernel 140
9.3.2 Experimente 143
9.3.2.1 Aufbau der Experimente 143
9.3.2.2 Messungen 144
9.3.2.3 Diskussion der Messergebnisse 148
9.4 Kernelfusion auf CPUs 152
9.4.1 Implementierung 152
9.4.1.1 Grundlegendes 152
9.4.1.2 Schleifenstrukturen für CPU-Kernel 152
9.4.1.3 Ausnutzung der Caches der CPU für die Wiederverwendung von Daten 154
9.4.1.4 Codegenerator für fusionierte CPU-Kernels 155
9.4.2 Experimente 158
9.4.2.1 Aufbau der Experimente 158
9.4.2.2 Messungen 159
9.4.2.3 Diskussion der Messergebnisse 165
9.5 Fazit 171

10 Tiling 173
10.1 Motivation 173
10.2 Theorie 173
10.2.1 Überblick 173
10.2.2 Blockweise Aufteilung von Systemkomponenten auf die parallelen Blöcke eines Kernels 175
10.2.3 Intervallbasierte Fusionsregeln 176
10.2.4 Fusion von (RHS | LC | MAP) $\rightarrow$ (LC | MAP | RED)-Abhängigkeiten zu einem Kernel 178
10.2.5 Unmöglichkeit der Fusion von LC $\rightarrow$ RHS-Abhängigkeiten zu einem Kernel 178

10.2.6 Partitionierte Fusion 178
10.2.6.1 Partitionierte Fusion vom Typ-PAIR: Fusion zu einem Kernelpaar . . 179
10.2.6.2 Partitionierte Fusion vom Typ-TRIP: Fusion zu einem Kerneltriplet . 181
10.2.7 Tiling Schemata 183
10.2.7.1 Grundidee 183
10.2.7.2 Trapezoidales Tiling-Schema 183
10.2.7.3 Hexagonales Tiling-Schema 185
10.2.8 Datenwiederverwendung 185
10.2.9 Tiling über die Stufen eines ODE-Verfahrens 186
10.2.10 Tiling über die Zeitschritte eines ODE-Verfahrens 188
10.2.10.1 Grundidee 188
10.2.10.2 Abrollen des Graphens 188
10.2.10.3 Zeit-Mapping-Distanz 191
10.2.10.4 Prolog- und Epilogiteration 193
10.2.11 Autotuning-Ansatz für die Parameter der Tiling-Schemata 193
10.2.11.1 Grundlegendes 193
10.2.11.2 Trapezoidales Tiling-Schema 194
10.2.11.3 Hexagonales Tiling-Schema 197
10.2.12 Abbildung der Tiles auf die Threads und die Kerne eines Prozessors . . . 200
10.2.12.1 Single-Thread-Tiling 200
10.2.12.2 Multithreaded-Single-Core-Tiling 200
10.2.12.3 Multi-Core-Tiling 201
10.2.12.4 Traditionelle Ansätze für das Tiling auf unterschiedlichen Architekturen 202
10.3 Tiling auf GPUs 203
10.3.1 Implementierung 203
10.3.1.1 Allgemeines 203
10.3.1.2 Fusion von LC $\rightarrow$ RHS-Abhängigkeiten 204
10.3.1.3 Datenwiederverwendung beim Tiling auf GPUs 207
10.3.1.4 Multi-Workgroup-Barrieren für das Multi-Workgroup-Tiling 208
10.3.1.5 Wahl des Mappings zwischen den Workitems und Vektorkomponenten 212
10.3.1.6 Deadlock-Freiheit für das Multi-Workgroup-Tiling 213
10.3.1.7 Überlegungen zur Tile-Breite und zum Autotuning 214
10.3.1.8 Codegenerierung 216
10.3.2 Experimente 218
10.3.2.1 Aufbau der Experimente 218
10.3.2.2 Messungen 219
10.3.2.3 Diskussion 232
10.4 Tiling auf CPUs 236
10.4.1 Implementierung 236
10.4.1.1 Allgemeines 236
10.4.1.2 Datentransfer zwischen Tiles und innerhalb eines Tiles 237
10.4.1.3 Schleifenstrukturen für das Single-Thread Tiling 239
10.4.1.4 Schleifenstrukturen für das Multi-Thread Tiling 246
10.4.1.5 Wahl des Mappings zwischen den Threads und Multi-Thread-Tiles . 248
10.4.1.6 Aliasing der temporären Arrays zur Reduktion des Workingsets . . . 251
10.4.1.7 Barrieren für das Multi-Thread-Tiling 251
10.4.1.8 Codegenerator für CPU-Kernels mit Tiling 251
10.4.2 Experimente 253

10.4.2.1 Aufbau der Experimente 253
10.4.2.2 Messungen 254
10.4.2.3 Diskussion 263
10.5 Fazit 269

11 Framework 271
11.1 Überblick 271
11.2 Workflow für einen Benutzer des Frameworks 271
11.3 Gliederung in Module 273
11.4 Unterstützte Zielplattformen und Erweiterung um neue Zielplattformen 274
11.5 Unterstützte ODE-Verfahrensklassen 274
11.6 Erweiterung des Frameworks um neue ODE-Verfahrensklassen 275
11.7 Modul des Datenflussgraphen 276
11.7.1 Allgemeines 276
11.7.2 Konstruktion eines Datenflussgraphen für eine neue Klasse von ODE-Verfahren 278
11.7.3 Anwendung der Enabling-Transformationen, und Erstellung der Varianten 279
11.7.4 Hilfsfunktionen für die Codegenerierung und die Ausführung 281
11.8 Modul der Codegeneratoren 281
11.9 Modul der Ausführung 282
11.9.1 Ablauf der Ausführung 282
11.9.2 Permanente Vektoren und deren Speichermanagement 283
11.9.3 Starten von Kernels 285
11.9.4 Unterstützung für NUMA-CPUs 286
11.10 Modul des Autotunings 287
11.11 Modul der DSL für ODE-Verfahren 288
11.11.1 Klassen-DSL 288
11.11.2 Verfahren-DSL 289
11.11.3 Implementierung der DSL 289

12 Schlussteil 292
12.1 Zusammenfassendes Fazit 292
12.1.1 Fazit zur Datenflussgraphrepräsentation 292
12.1.2 Fazit zur Basisimplementierung 292
12.1.3 Fazit zur Kernelfusion 292
12.1.4 Fazit zum Tiling 293
12.1.5 Fazit zum Framework 294
12.1.6 Abschließendes Fazit 295
12.2 Erfüllung der Ziele 295
12.3 Ausblick 296

13 Anhang 300
13.1 GPU-APIs mit CPU-Unterstützung 300
13.2 Alternative Möglichkeiten zur Integration der CPU-Unterstützung in unser Framework 300
13.3 Herausforderungen bei den experimentellen Untersuchungen 302
13.3.1 Workaround für die Untersuchung der Skalierbarkeit mit der Anzahl an benutzten GPU-Kernen 302

13.3.2 Probleme bei den Experimenten für Multi-Workgroup-Tiling auf der Polaris-GPU . . . 302
13.4 Metriken . . . 304
13.5 GPU-Daten . . . 307
13.6 CPU-Daten . . . 314
13.7 Verwendete RK-Verfahren . . . 319

14 Literaturverzeichnis 322
14.1 Eigene Publikationen . . . 322
14.2 Artikel aus wissenschaftlichen Zeitschriften . . . 324
14.3 Beiträge aus wissenschaftlichen Konferenzen . . . 329
14.4 Bücher . . . 335
14.5 Technische Spezifikationen, Dokumentationen, Berichte und Whitepaper . . . 337
14.6 Internetseiten . . . 341

15 Eidesstattliche Versicherung 343

ABBILDUNGSVERZEICHNIS

3.1 Schematisches Blockdiagramm einer modernen GPU . . . 41
3.2 Schematisches Blockdiagramm eines Kerns einer modernen GPU . . . 42
3.3 Anordnung von Datentypen unterschiedlicher Breite in dem Vektorregistersatz einer GPU . . . 44
3.4 Iterationsraum eines 2D-Kernels . . . 47
3.5 Vektorisierung von arithmetischen Operationen in einem Kernel . . . 48
3.6 Vektorisierung eines if-else-Blocks in einem Kernel . . . 49
3.7 Vektorisierung einer while-Schleife in einem Kernel . . . 49
3.8 Zugriff auf den Arbeitsspeicher mit einer SIMD-Blockleseinstruktion . . . 53
3.9 Zugriff auf den Arbeitsspeicher mit einer Gather-Instruktion . . . 53
3.10 Vektorisierung eines Speicherzugriffs mit einer Gather-Instruktion in einem Kernel 54

5.1 Klassifikation der räumlichen Strukturen von Anfangswertproblemen . . . 76
5.2 Zugriffsmuster einer RHS-Funktion für allgemeine Anfangswertprobleme . . . 80
5.3 Zugriffsmuster einer RHS-Funktion für Anfangswertprobleme mit beschränkter Zugriffsdistanz . . . 80

6.1 Datenflussgraph des Heun-Euler-Verfahrens . . . 87
6.2 Datenflussgraph des Midpoint-Verfahrens mit annotierten Zeitschrittdistanzen . . 89
6.3 Datenflussgraph eines Adams-Bashforth-Verfahrens mit annotierten Zeitschrittdistanzen . . . 89
6.4 Die Datenflussgraphen vom expliziten und impliziten Euler-Verfahren . . . 90
6.5 RHS → LC-Abhängigkeitskette aus RHS → LC-Gliedern . . . 93
6.6 Die Abhängigkeiten zwischen den Systemkomponenten in einer RHS → LC-Abhängigkeitskette . . . 93
6.7 Datenflussgraph eines vierstufigen RK-Verfahrens mit zwei unabhängigen Stufen 94
6.8 Datenflussgraph eines PIRK-Verfahrens . . . 95

8.1 Pseudocode der GPU-Kernel der Basisimplementierung . . . 101
8.2 Pseudocode der CPU-Kernel der Basisimplementierung . . . 106

9.1 Datentransfer in einem fusionierten Kernel . . . 118
9.2 Vermeidung von redundanten Ladeoperationen in einem fusionierten Kernel . . . 118
9.3 Reduktion der Anzahl an zurückgeschriebenen Vektoren durch Kernelfusion . . 119
9.4 Mögliche Fusionen in der Abhängigkeitskette . . . 121
9.5 Cloning-Transformation . . . 122
9.6 Splitting-Transformation . . . 123
9.7 Move-Transformation . . . 124
9.8 Datenflussgraphen der Varianten des PIRK-Verfahrens . . . 133
9.9 Datenflussgraphen der Varianten der Peer-Verfahren. . . . 136
9.10 Schematisierte Beispiele für den generierter GPU-Code für die fusionierten Varianten . . . 142
9.11 Skalierung der Laufzeit mit der Gittergröße für die Varianten des Verner-Verfahrens auf Maxwell . . . 146
9.12 Skalierung der Varianten mit Kernelfusion mit der Anzahl der verwendeten GPU-Kerne. . . . 147

9.13 Implementierte Schleifenstrukturen für ein fusioniertes CPU-Kernel. 153
9.14 Pseudocode für die Datenwiederverwendung per Array-Contraction und für das Zurückschreiben per Non-Temporal-Stores 156
9.15 Skalierung des Durchsatzes mit der Gittergröße für die Varianten des Verner-Verfahrens und des PIRK-Verfahrens mit Non-Temporal-Store-Instruktionen auf Skylake 162
9.16 Skalierung des Durchsatzes mit der Gittergröße für die Varianten des Verner-Verfahrens und des PIRK-Verfahrens mit regulären Store-Instruktionen auf Skylake 163
9.17 Skalierung der Varianten mit Kernelfusion mit der Anzahl der verwendeten CPU-Kerne. 164

10.1 Partitionierte Fusion einer LC → RHS-Abhängigkeit vom Typ-PAIR 179
10.2 Partitionierte Fusion einer LC → RHS-Abhängigkeit vom Typ-TRIP 181
10.3 Trapezoidales Tiling-Schema 184
10.4 Hexagonales Tiling-Schema 184
10.5 Zwischen zwei Kernels über den DRAM transferierte Vektorkomponenten 187
10.6 Tiling über die Zeitschritte 189
10.7 Beispiel für die Anwendung der Abrolltransformation auf das explizite Eulerverfahren 190
10.8 Single-Workgroup-Tiling 204
10.9 Multi-Workgroup-Tiling 204
10.10 Datenwiederverwendung in einem Tile auf der GPU 209
10.11 Mikro-Benchmark der Multi-Workgroup-Barrieren auf einer Volta-GPU 211
10.12 Implementierte Strides beim Tiling auf GPUs 214
10.13 Generierter Code für ein GPU-Kernel mit Tiling. 217
10.14 Laufzeituntersuchungen für das Verner-Verfahren mit Tiling auf Volta 221
10.15 Laufzeituntersuchungen für das Bogacki-Shampine-Verfahren mit Tiling auf Volta 222
10.16 Laufzeituntersuchungen für das Verner-Verfahren mit Tiling auf Kepler 223
10.17 Laufzeituntersuchungen für das Verner-Verfahren mit Tiling auf Polaris 224
10.18 Untersuchungen zur besten Tile-Breite und der besten Anzahl an nebenläufigen Tiles beim Multi-Workgroup-Tiling 225
10.19 Einfluss der Blöcke per Thread und der Strides beim Multi-Workgroup-Tiling auf die Performance 226
10.20 Untersuchungen zur besten Tile-Höhe beim Multi-Workgroup-Tiling 226
10.21 Profiling des Tilings auf Volta 227
10.22 Profiling des Tilings auf Kepler 228
10.23 Laufzeituntersuchungen für das Euler-Verfahren mit Single-Workgroup-Tiling auf Kepler 229
10.24 Laufzeituntersuchungen für das Heun-Verfahren mit Single-Workgroup-Tiling auf Kepler 230
10.25 Laufzeituntersuchungen für das RK-3/8-Verfahren mit Single-Workgroup-Tiling auf Kepler 231
10.26 Abarbeitung eines Single-Thread-Tiles 237
10.27 Abarbeitung eines Multi-Thread-Tiles 237
10.28 Lesen und Zurückschreiben aus beziehungsweise in permanente Vektoren via explizite Kopierschleifen. 240
10.29 Lesen und Zurückschreiben aus beziehungsweise in permanente Vektoren via Loop-Splitting. 241

10.30 Lesen und Zurückschreiben aus beziehungsweise in permanente Vektoren via Index-basierten bedingten Anweisungen. 242
10.31 Pseudocode der Simple-Tiled-Schleifenstrukturen des Single-Thread-Tilings auf CPUs 244
10.32 Pseudocode der In-Link-Subblocked-Schleifenstrukturen des Single-Thread-Tilings auf CPUs 245
10.33 Pseudocode der Schleifenstrukturen mit Blocke-Wise-Mapping für das Multi-Thread-Tiling auf CPUs 249
10.34 Pseudocode der Schleifenstrukturen mit Blocke-Cyclic-Mapping für das Multi-Thread-Tiling auf CPUs. 250
10.35 Laufzeituntersuchungen für das Verner-Verfahren mit Tiling auf Skylake 257
10.36 Laufzeituntersuchungen für das Bogacki-Shampine-Verfahren mit Tiling auf Skylake 258
10.37 Laufzeituntersuchungen für das Verner-Verfahren mit Tiling auf Ryzen 259
10.38 Untersuchungen zur besten Tile-Breite und der besten Anzahl an nebenläufigen Tiles beim Multi-Thread-Tiling 260
10.39 Untersuchungen zur besten Tile-Höhe beim Multi-Thread-Tiling 261
10.40 Profiling des Multi-Thread-Tilings auf Skylake 262

11.1 Schematischer Arbeitsfluss für die Verwendung unseres Frameworks durch einen Benutzer 272
11.2 Module und deren Abhängigkeiten in unserem Framework. 273
11.3 Konstantenklasse für die RK-Verfahren 277
11.4 Funktionen für das Hinzufügen von Grundoperationen zu einem Datenflussgraphen 280
11.5 Funktionen für das Hinzufügen von Abhängigkeiten zwischen Grundoperationen 280
11.6 Konstruktionsmethode für den Datenflussgraph für ein Adams-Bashforth-Verfahren 280
11.7 Vorgehen der Ausführung unseres Frameworks 284
11.8 Blockweise Aufteilung der Vektorkomponenten auf die NUMA-Domains einer NUMA-CPU 287
11.9 Aufteilung der Tiles auf die NUMA-Domains einer NUMA-CPU 287
11.10 Codebeispiel für die Klassen-DSL 290
11.11 Codebeispiel für die Verfahren-DSL 290

Tabellenverzeichnis

3.1 Zentrale Begriffe der GPU-Terminologien unterschiedlicher Hersteller 38
3.2 Anzahl an simultanen ausgeführten Threads pro Subkern in Abhängigkeit von der Anzahl an Registern pro Thread auf einer Volta-GPU 56

6.1 Mögliche Abhängigkeiten in dem Datenflussgraphen eines ODE-Verfahrens . . . 88

8.1 Laufzeitmessung und Profiling der GPU-Kernel der Basisimplementierung . . . 103
8.2 Laufzeitmessung und Profiling der CPU-Kernel der Basisimplementierung . . . 108

9.1 Speicherzugriffmuster für die Grundoperationen eines Blocks eines Kernels . . . 115
9.2 Fusionierbarkeit zweier Vektorgrundoperationen bei einer direkten Abhängigkeit. 116
9.3 Kernel der Implementierungsvarianten des Verner-Verfahrens 127
9.4 Kernel der Implementierungsvarianten des Verner-Verfahrens (Fortsetzung) . . . 128
9.5 Zahl der gelesenen und geschriebenen Vektoren für das Verner-Verfahren 128
9.6 Kernel der Implementierungsvarianten des DOPRI-Verfahrens 130
9.7 Kernel der DOPRI-ODEINT-Variante 131
9.8 Zahl der gelesenen und geschriebenen Vektoren für das DOPRI-Verfahren. . . . 131
9.9 Kernel der Implementierungsvarianten der PIRK-Verfahren 134
9.10 Zahl der gelesenen und geschriebenen Vektoren für das PIRK-Verfahren 134
9.11 Kernel der Implementierungsvarianten der Peer-Verfahren 137
9.12 Zahl der gelesenen und geschriebenen Vektoren der Peer-Verfahren 137
9.13 Laufzeitmessungen für die Varianten mit Kernelfusion auf GPUs 145
9.14 Speedup für die Varianten mit Kernelfusion auf GPUs 145
9.15 Vergleich mit DOPRI-ODEINT auf der Maxwell 145
9.16 Profiling der Varianten mit Kernelfusion auf Maxwell 146
9.17 Messung von Laufzeit und Speedup für die Varianten mit Kernelfusion auf CPUs 161
9.18 Profiling der Varianten mit Kernelfusion auf dem Skylake 161

10.1 Eingabe- und Ausgabeintervalle der Vektorgrundoperationen für ein gegebenes Argumentintervall. 176
10.2 Blockweise Aufteilung von Vektorgrundoperation auf die Blöcke eines Kernels . 176
10.3 Blockzyklische Datenverteilung bei einer partitionierten Fusion vom Typ-PAIR . 180
10.4 Blockzyklische Datenverteilung bei einer verteilten Fusion vom Typ-TRIP. . . . 182

13.1 Technische Daten dreier NVIDIA GPUs aus den Jahren 2014 bis 2016 309
13.2 Technische Daten dreier NVIDIA GPUs aus den Jahren 2017 bis 2020 310
13.3 Technische Daten dreier AMD GPUs aus den Jahren 2014 bis 2020 311
13.4 Technische Daten dreier Intel GPUs aus den Jahren 2013 bis 2020 312
13.5 Rohperformance der GPUs aus den Jahren 2014 bis 2020 313
13.6 Technische Daten dreier Intel CPUs aus den Jahren 2012 bis 2019 316
13.7 Technische Daten dreier AMD CPUs aus den Jahren 2013 bis 2021 317
13.8 Rohperformance der CPUs aus den Jahren von 2014 bis 2020 318

1 Einleitung

1.1 Motivation

Viele wissenschaftliche Simulationen verwenden Systeme von gewöhnlichen Differentialgleichungen als mathematische Modelle, um Phänomene der echten Welt zu beschreiben. Die folgende Arbeit betrachtet *Anfangswertprobleme* (englisch: initial value problems, abgekürzt *IVPs*) von Systemen von *gewöhnlichen Differentialgleichungen* (englisch: *ordinary differential equations*, abgekürzt: *ODEs*):

$$\textbf{Gegeben:}\quad \mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad t_0, \quad \mathbf{y}(t_0) = \mathbf{y}_0, \quad t_e \tag{1.1}$$

$$\textbf{Gesucht:}\quad \mathbf{y}_e = \mathbf{y}(t_e) \tag{1.2}$$

Bei einem ODE-System bestehend aus n Gleichungen gilt, dass $\mathbf{y} \in \mathbb{R}^n$, $t \in \mathbb{R}$ sowie $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ ist. Für das Lösen von Anfangswertproblemen wurde eine Vielzahl von *ODE-Verfahren* entwickelt. Der klassische numerische Lösungsansatz, welcher auch von den ODE-Verfahren in dieser Arbeit angewendet wird, besteht aus einer Zeitschrittprozedur. Diese Prozedur beginnt zur Simulationszeit t_0 mit einem Anfangswert $\mathbf{y}_0$ und führt eine Folge von Zeitschritten $t_\kappa \to t_{\kappa+1}$ mit $\kappa = 0, 1, 2, \ldots$ aus, bis sie die finale Zeit t_e der Simulation erreicht hat. Bei jedem Zeitschritt κ wendet die Zeitschrittprozedur die *Funktion der rechten Seite* (englisch: *right-hand-side*, abgekürzt: *RHS*) an, um einen neuen Zustand der Simulation $\mathbf{y}_{\kappa+1}$ zu berechnen, der die exakte Lösung der Funktion $\mathbf{y}(t)$ zum Zeitpunkt $t_{\kappa+1}$ approximiert.

Das Lösen solcher Anfangswertprobleme ist nicht nur für die mathematische, physikalische, chemische, biologische sowie medizinische Forschung, sondern auch für ingenieurtechnische Anwendungen von großer Bedeutung. Auch müssen solche Anfangswertprobleme für die Spiellogik und die Graphik von modernen Computerspielen sowie, um CGI-Spezialeffekte in Filmen zu generieren, gelöst werden. Für viele solcher Anwendungen muss zudem zwingend ein Anfangswertproblem mit einer großen Problemgröße, das heißt ein ODE-System bestehend aus sehr vielen Gleichungen, gelöst werden. Für viele Anwendungen gilt zudem, dass je größer das ODE-System gewählt wird, umso besser wird auch die Genauigkeit beziehungsweise die Qualität des Endergebnisses.

Für das Lösen von Anfangswertproblemen stehen eine Vielzahl von unterschiedlichen ODE-Verfahren zur Verfügung, die sich sowohl in ihrer Berechnungsstruktur, Rechenaufwand, Genauigkeit beziehungsweise Ordnung, und ihrer Eignung für ein spezifisches Anfangswertproblem unterscheiden. In dieser Arbeit betrachten wir mehrere Vertreter der Klasse der *expliziten allgemeinen linearen Verfahren*, und zwar die *expliziten RK-Verfahren*, die *PIRK-Verfahren*, die *Peer-Verfahren* und die *Adams-Bashforth-Verfahren*. Bei all diesen Verfahren setzt sich ein Zeitschritt aus Auswertungen der rechten Seite (*RHS-Operationen*), Linear-Kombinationen (*LC-Operationen*), Map-Funktionen (*MAP-Operationen*) und Reduktionen (*RED-Operationen*) zusammen. In diesen ODE-Verfahren können drei Arten von Parallelität auftreten: Eine Parallelität über die Komponenten des ODE-Systems (*Systemparallelität*), eine Parallelität über die Grundoperationen des ODE-Verfahrens, welche voneinander unabhängig sind, (*Methodenparallelität*) und eine Parallelität über die Zeitschritte des ODE-Verfahrens (*Zeitparallelität*). Bei LC-, MAP- und RED-Operationen ist die Systemparallelität immer vorhanden und beinhaltet auch einen sehr hohen Grad an Datenparallelität. Dahingegen ist die Systemparallelität und deren Datenparallelität bei RHS-Operationen beziehungsweise bei RHS-Funktionen vom zu lösenden Anfangswertproblem abhängig. Allerdings besitzen die RHS-Funktionen von den

allermeisten relevanten Anfangswertproblemen einen sehr hohen Grad an Systemparallelität sowie Datenparallelität entlang der Systemdimension.

Da viele Anwendungen Anfangswertprobleme mit sehr vielen oder mit möglichst vielen Systemkomponenten lösen müssen, wodurch das Lösen dieser Anfangswertprobleme wiederum sehr rechenintensiv ist, ist es vielversprechend, das Lösen von Anfangswertproblemen durch optimierte Lösungsprogramme zu beschleunigen. Da ein Lösen von extrem großen Anfangswertproblemen mit einem ODE-Verfahren auf Grund der Systemparallelität nicht nur massiv parallel, sondern auch massiv datenparallel ist, eignen sich moderne SIMD-Parallelrechner wie Mehrkern-CPUs und GPUs hervorragend, um ein solches Lösungsprogramm mit einem ODE-Verfahren effizient auszuführen.

Für *dichtbesetzte Anfangswertprobleme*, das heißt für Anfangswertprobleme, bei welchen die RHS-Funktion zur Bestimmung der zeitlichen Ableitung einer Komponente die meisten aller anderen Komponenten des Argumentvektors betrachten muss, dominieren die RHS-Operationen die Laufzeit eines jeden Lösungsprogramms für Anfangswertprobleme. Besitzt bei einem dichtbesetzten Anfangswertproblem die RHS-Funktion zusätzlich eine *hohe arithmetische Intensität*, dann kann ein solches Lösungsprogramm für dichtbesetzte Anfangswertprobleme die Rechenleistung eines Prozessors leicht gut ausnutzen. Für *dünnbesetzte Anfangswertprobleme*, bei welchen die RHS-Funktion zur Bestimmung der zeitlichen Ableitung einer Komponente nur wenige andere Komponenten des Argumentvektors betrachten muss, besitzen die RHS-Funktion beziehungsweise die RHS-Operationen zusätzlich oft eine *niedrige arithmetische Intensität*. Da alle anderen Operationen der betrachteten ODE-Verfahren, und zwar LC-Operationen, MAP-Operationen sowie RED-Operationen, ebenfalls eine niedrige arithmetische Intensität besitzen, ist für diese ODE-Verfahren das Lösen eines dünnbesetzten Anfangswertproblems mit einer niedrigen arithmetischen Intensität durch ein naives Lösungsprogramm ohne Lokalitätsoptimierungen sehr stark durch die Speicherbandbreite beziehungsweise durch die DRAM-Bandbreite gebunden. Deshalb ist es für solche dünnbesetzten Anfangswertprobleme mit einer niedrigen arithmetischen Intensität zur Verbesserung der Performance vielversprechend, ein Lösungsprogramm mit Lokalitätsoptimierungen zu verwenden. Allerdings verhindert der allgemeine Fall eines Anfangswertproblems, bei welchem die RHS-Funktion für die Auswertung einer Komponente auf beliebige andere Komponenten im Argumentvektor zugreifen darf, viele Lokalitätsoptimierungen. Dies lässt sich mit einer Spezialisierung auf bestimmte Anfangswertprobleme vermeiden, welche das Zugriffsmuster der RHS-Funktion auf ihren Argumentvektor einschränken. Eine vielversprechende solche Einschränkung, welche wir in dieser Arbeit für zusätzliche Lokalitätsoptimierungen verwenden, ist die sogenannte *beschränkte Zugriffsdistanz*. Bei Anfangswertproblemen mit einer beschränkten Zugriffsdistanz greift die RHS-Funktionen für die Auswertung einer Komponente nur auf diejenigen Komponenten im Argumentvektor zu, welche sich innerhalb der Zugriffsdistanz um der auszuwertenden Komponente herum befinden. Somit deckt die beschränkte Zugriffsdistanz viele weit verbreitete Anfangswertprobleme wie Stencils ab.

Zusätzlich existieren eine große Anzahl an relevanten Anfangswertproblemen, ODE-Verfahren und Hardwarearchitekturen, wobei potentiell jede Kombination von Anfangswertproblem, ODE-Verfahren und Hardwarearchitektur von einer speziell optimierten Implementierungsvariante profitiert. Deshalb ist es sehr zeitaufwändig, per Hand eine optimierte Implementierungsvariante für jede dieser Kombinationen zu erstellen. Eine solche optimierte Implementierungsvariante ist jedoch erforderlich, sofern ein Benutzer die Rechenleistung eines Rechners für das Lösen eines Anfangswertproblems effizient ausnutzen möchte.

1.2 Zielstellung

Aus den soeben genannten Gründen stellen wir uns für diese Arbeit als Ziel, einen ganzheitlichen Ansatz zum automatischen Generieren von Implementierungsvarianten von expliziten ODE-Verfahren für das Lösen von Anfangswertproblemen zu entwickeln:

- für ein breites Spektrum an häufig verwendeten, expliziten ODE-Verfahren,
- für das Lösen von Anfangswertproblemen ohne Einschränkungen des möglichen Zugriffsmusters,
- für das Lösen von Anfangswertproblemen mit beschränkter Zugriffsdistanz,
- mit dem Ausnutzen von Systemparallelität und Methodenparallelität,
- mit einem hohen Grad an Datenwiederverwendung über die On-Chip-Speicher, und
- mit GPUs und CPUs als optimierte Zielplattformen.

1.3 Vorgehen

1.3.1 Abstraktion von ODE-Verfahren durch einen Datenflussgraphen

Für die automatische Generierung von Implementierungsvarianten für ein breites Spektrum an häufig verwendeten, expliziten ODE-Verfahren müssen wir zuerst eine gemeinsame Abstraktion für all diese Verfahren finden. In dieser Arbeit entschieden wir uns für eine *Datenflussgraphrepräsentation*, welche den Zeitschritt eines ODE-Verfahrens durch einen Datenflussgraphen beschreibt. Ein solcher Datenflussgraph besteht im Wesentlichen aus Auswertungen der rechten Seite (*RHS-Operationen*), Linearkombinationen (*LC-Operationen*), Map-Funktionen (*MAP-Operationen*) und Reduktionen (*RED-Operationen*) als Knoten beziehungsweise als *Grundoperationen*. Hierdurch können wir die Klasse der allgemeinen linearen Verfahren und dadurch die meisten häufig verwendeten expliziten ODE-Verfahren abdecken. Insbesondere decken wir dadurch die RK-Verfahren, die PIRK-Verfahren, die Peer-Verfahren und die Adams-Bashforth-Verfahren ab. Für viele ODE-Verfahren ergibt die *transitive Reduktion* des Datenflussgraphen, also die Entfernung sämtlicher Abhängigkeiten, die die topologische Sortierung nicht beeinflussen, eine *RHS → LC-Abhängigkeitskette*, die sich aus *RHS → LC-Gliedern* zusammensetzt. Die LC-Operationen in dieser Kette verwenden nicht nur die RHS-Operation ihres jeweiligen RHS → LC-Glieds als Argumentvektor, sondern sie können beliebig viele RHS-Operationen oder LC-Operationen aus vorherigen RHS → LC-Gliedern als Argumentvektoren verwenden.

1.3.2 Erzeugung der Basisimplementierungen aus dem Datenflussgraphen eines ODE-Verfahrens

Anhand eines solchen Datenflussgraphen eines ODE-Verfahrens können wir bereits leicht automatisch eine *Basisimplementierung* erstellen, die die Grundoperationen im Datenflussgraphen gemäß ihrer topologischen Sortierung ausführt. Innerhalb jeder Grundoperation kann die Basisimplementierung die Parallelität über die Systemdimension ausnutzen, um die Berechnungen der Grundoperation auf mehrere Kerne beziehungsweise Threads des verwendeten Prozessors zu verteilen. Somit erzeugt die Basisimplementierung für jede RHS-Operationen ein paralleles

RHS-Kernel, für jede LC-Operationen ein paralleles *LC-Kernel*, für jede MAP-Operationen ein paralleles *MAP-Kernel* und für jede RED-Operationen ein paralleles *RED-Kernel*. Diese Basisimplementierung kann bereits bei dichtbesetzten Anfangswertproblemen eine gute Performance erzielen, da bei diesen Anfangswertproblemen die RHS-Operationen eine quadratische Komplexität bezüglich der Systemgröße besitzen, die Laufzeit dominieren und zudem, falls sie eine hohe arithmetische Intensität besitzen, durch die Rechenleistung des Prozessors gebunden sind. Jedoch sind bei dünnbesetzten Anfangswertproblemen mit einer niedrigen arithmetischen Intensität, bei welchen die RHS-Operationen eine lineare Komplexität bezüglich der Systemgröße besitzen, sämtliche Grundoperationen, also die RHS-, MAP-, LC- und RED-Operationen, durch die DRAM-Bandbreite gebunden. Deshalb sind bei solchen dünnbesetzten Anfangswertproblemen mit einer niedrigen arithmetischen Intensität zusätzliche Lokalitätsoptimierungen vielversprechend, um die Performance deutlich zu erhöhen.

1.3.3 Erhöhung der Datenwiederverwendung durch die Kernelfusion für allgemeine Anfangswertprobleme

Um die Datenwiederverwendung für allgemeine Anfangswertprobleme ohne Einschränkungen des möglichen Zugriffsmusters zu erhöhen, entschieden wir uns für die Technik der *Kernelfusion*. Bei dieser Technik werden die Grundoperationen im Datenflussgraphen zu größeren *Kernels* fusioniert. Innerhalb eines Kernels kann ein Prozessor dann seine On-Chip-Speicher ausnutzen, um die produzierten Daten wiederzuverwenden, sowie ein mehrfaches Lesen von den benötigten Daten aus dem DRAM und ein unnötiges Zurückschreiben von den produzierten Daten in den DRAM zu vermeiden. Ebenso kann ein fusioniertes Kernel die Systemparallelität der fusionierten Grundoperationen weiterhin dafür ausnutzen, um seine Berechnungen seiner fusionierten Grundoperationen auf sämtlichen Rechenkernen eines Prozessors durchzuführen. Allerdings kann die Kernelfusion die Grundoperationen im Datenflussgraphen nicht beliebig verschmelzen. Insbesondere sind LC $\rightarrow$ RHS-Abhängigkeiten unverschmelzbar, da im Falle eines allgemeinen Anfangswertproblems die RHS-Funktion für die Auswertung einer ihrer Komponenten beliebig auf ihren Argumentvektor zugreifen darf. Dadurch werden diese LC $\rightarrow$ RHS-Abhängigkeiten zu globalen Barrieren. Deshalb darf die Kernelfusion zwar sämtliche Grundoperationen innerhalb eines RHS $\rightarrow$ LC-Glieds miteinander verschmelzen, jedoch darf sie nicht die Grundoperationen aus unterschiedlichen RHS $\rightarrow$ LC-Gliedern miteinander verschmelzen. Zusätzlich kann die Kernelfusion voneinander unabhängige Grundoperationen verschmelzen, wie sie in den unabhängigen Stufen eines ODE-Verfahrens auftreten. Hierdurch können wir mit der Kernelfusion die Methodenparallelität des ODE-Verfahrens für die Datenwiederverwendung ausnutzen.

Um weitere Fusionen zu ermöglichen und um die Datenwiederverwendung weiter zu erhöhen, wenden wir zusätzlich vor der Kernelfusion mehrere *Enabling-Transformationen* auf den Datenflussgraphen an. Hierunter fallen die *Splitting-Transformation* zum Aufspalten von LC-Operationen und die *Cloning-Transformation* zur redundanten Berechnung von Grundoperationen. Ebenso entwickeln wir ein einfaches *Performance-Modell*, das über das *Datenvolumen* die Performance der fusionierten Kernels abschätzt. Dabei beschreibt dieses Datenvolumen, welche Datenmenge die Kernels aus dem DRAM laden und in den DRAM schreiben müssen, unter der Annahme, dass innerhalb eines Kernels die Caches des Prozessors mehrmalige Zugriffe auf dieselbe Speicheradresse immer abfangen, und dass es keine Datenwiederverwendung zwischen zwei aufeinanderfolgenden Kernels über die Caches des Prozessors gibt. Anschließend bewerten wir mit diesem Performance-Modell mehrere generierte Varianten für zwei explizi-

te RK-Verfahren, und zwar das *Verner-Verfahren* und das *DOPRI-Verfahren*, sowie mehrere generierte Varianten für die Peer-Verfahren und die PIRK-Verfahren.

Die so fusionierten Grundoperationen in dem Datenflussgraphen eines ODE-Verfahrens beziehungsweise die fusionierten Kernels lassen sich nun mit Hilfe von Codegeneratoren zu *GPU-Kernels* beziehungsweise *CPU-Kernels* umsetzen. Für beide Zielarchitekturen optimieren wir den generierten Code so, dass er die On-Chip-Speicher der jeweiligen Architektur effizient zur Datenwiederverwendung ausnutzt. Unsere experimentellen Untersuchungen für die per Kernelfusion generierten Implementierungsvarianten zeigen, dass sowohl die Kernelfusion als auch die Enabling-Transformationen zwar die Datentransfers zwischen GPU und DRAM beziehungsweise zwischen CPU und DRAM gegenüber der Basisimplementierung deutlich reduzieren und dadurch die Performance deutlich verbessern können. Jedoch zeigen unsere Untersuchungen auch, dass die Performance der per Kernelfusion generierten Implementierungsvarianten weiterhin sehr stark durch die DRAM-Bandbreite gebunden ist. Jedoch sind wir nun bei den allgemeinen Anfangswertproblemen ohne Einschränkungen des möglichen Zugriffsmusters an die Grenzen des Machbaren gestoßen, da das beliebige Zugriffsmuster der RHS-Funktion weitere Lokalitätsoptimierungen, das heißt die Datenwiederverwendung über zwei unterschiedliche RHS $\rightarrow$ LC-Glieder hinweg, verhindert.

1.3.4 Erhöhung der Zugriffslokalität durch Tiling für Anfangswertprobleme mit beschränkter Zugriffsdistanz

Wir umgehen diese Beschränkungen durch eine Spezialisierung auf Anfangswertprobleme mit *beschränkter Zugriffsdistanz*, wie sie zum Beispiel bei Stencil-Problemen auftreten. Bei solchen Anfangswertproblemen muss die RHS-Funktion für die Auswertung einer Komponente nicht auf den kompletten Argumentvektor zugreifen, sondern nur auf diejenigen Komponenten des Argumentvektors, die sich innerhalb der Zugriffsdistanz um der auszuwertenden Komponente herum befinden. Aber selbst bei solchen Anfangswertproblemen sind LC $\rightarrow$ RHS-Abhängigkeiten nicht direkt zu einem einzigen Kernel verschmelzbar, sondern sie sind nur über eine partitionierte Fusion zu einem Kernelpaar oder zu einem Kerneltriplet verschmelzbar.

Mit dieser partitionierten Fusion lassen sich zweidimensionale *Tiling-Schemata* verwirklichen, welche den Iterationsraum, der von der Systemdimension als räumliche Dimension und der RHS $\rightarrow$ LC-Abhängigkeitskette als zeitliche Dimension aufgespannt wird, mit *Tiles* (Kacheln) auffüllen. Als zweidimensionale Tiling-Schemata realisieren wir das *trapezoidale Tiling-Schema* und das *hexagonale Tiling-Schema*, welche den Iterationsraum mit Trapezoiden beziehungsweise mit Hexagonen als Tiles auffüllen. Bei diesen beiden Tiling-Schemata verläuft die *Breite* eines Tiles entlang der Systemdimension, während die *Höhe* eines Tiles entlang der zeitlichen Dimension beziehungsweise entlang der RHS $\rightarrow$ LC-Abhängigkeitskette verläuft. In einem solchen Tile kann nun ein Prozessor wieder seine On-Chip-Speicher für die Datenwiederverwendung ausnutzen, wodurch das Tiling die Zahl der Vektorkomponenten, die ein Prozessor aus dem DRAM laden oder in den DRAM schreiben muss, reduzieren kann. Zusätzlich kann das Tiling weiterhin die Systemparallelität ausnutzen, indem es voneinander unabhängige Tiles parallel zueinander von den Rechenkernen des Prozessors berechnen lässt.

Wir können mit den beiden Tiling-Schemata nicht nur ein *Tiling über die Stufen*, bei welchem das Tiling mit dem ersten RHS $\rightarrow$ LC-Glied der Abhängigkeitskette anfängt und mit dem letzten RHS $\rightarrow$ LC-Glied der Abhängigkeitskette aufhört, durchführen, sondern auch ein *Tiling über die Zeitschritte*, bei welchem das Tiling über die Zeitschrittgrenzen fortgesetzt wird.

Um dieses Tiling über die Zeitschritte zu definieren, stellen wiederum erweiterte Regeln für das Tiling vor. Ebenso müssen wir für das Tiling über die Zeitschritte den Datenflussgraphen eventuell durch eine besondere *Abroll-Transformation* abrollen, so dass danach die Grundoperationen in dem abgerollten Datenflussgraphen nicht mehr nur einen einzigen, sondern mehrere aufeinanderfolgende Zeitschritte des ODE-Verfahrens verkörpern. Ebenso benötigt das Tiling über die Zeitschritte, da dessen Tiles nun keinen glatten Schnitt mehr an den Zeitschrittgrenzen besitzen, zur Initialisierung vor dem ersten Zeitschritt eine spezielle *Prolog-Iteration* und nach dem letzten Zeitschritt zum Abschluss eine spezielle *Epilog-Iteration*. Zusätzlich entwickeln wir für sowohl für das trapezoidale als auch für das hexagonale Tiling-Schema einen Autotuning-Ansatz, welcher für diese beiden Tiling-Schemata per Autotuning die optimalen Parameter, welche im Wesentlichen die Tile-Höhen und die Tile-Breiten sind, bestimmt.

In der einfachsten Implementierung des Tilings wird ein Tile nur von einem einzelnen Kern beziehungsweise einem einzelnen Thread bearbeitet. Diese Art des Tilings bezeichnen wir als *Single-Core-Tiling*. Bei dem Single-Thread-Tiling benötigt ein moderner Mehrkernprozessor wiederum viele Tiles, um ausgelastet zu sein, wodurch jedes Tile nur einen kleinen Anteil des On-Chip-Speichers des Prozessors belegen kann und die maximalen Tile-Größe, die noch im On-Chip-Speicher des Prozessors Platz findet, stark begrenzt ist. Jedoch sind beim Tiling potentiell größere Tiles vorteilhaft, da diese das Datenvolumen stärker reduzieren können als kleinere Tiles. Wir umgehen dieses Problem, indem wir mehrere Kerne beziehungsweise mehrere Threads an einem Tile zusammenarbeiten lassen, wodurch der Prozessor wiederum weniger Tiles für seine Auslastung gleichzeitig berechnen muss. Diese Art des Tiling bezeichnen wir als *Multi-Core-Tiling*.

Aus dem Datenflussgraphen eines ODE-Verfahrens und einem Tiling lassen sich nun wieder mit Hilfe eines Codegenerators optimierte Implementierungsvarianten für beide Zielplattformen, also für GPUs und CPUs, erstellen, die die Ausnutzung des On-Chip-Speichers des jeweiligen Prozessors maximieren. Unsere experimentellen Untersuchungen mit dem Tiling zeigen, dass sowohl auf CPUs als auch auf GPUs das Tiling die Datentransfers zwischen dem Prozessor und dem DRAM stark reduzieren kann. Darüber lässt sich mit dem Tiling gegenüber den Varianten mit Fusion oder der Basisimplementierung ein deutlicher Speedup erzielen. Ebenso zeigen unsere Untersuchungen, dass sich das Single-Core-Tiling gegenüber dem Multi-Core-Tiling oder einer per Kernelfusion generierten Variante nur für kleine Zugriffsdistanzen lohnt, während das Multi-Core-Tiling noch für sehr große Zugriffsdistanzen gegenüber einer per Kernelfusion generierten Variante einen Speedup erzielen kann.

1.3.5 Implementierung unseres Ansatzes in einem Framework

Wir entwickelten für die Umsetzung unseres Ansatzes, das heißt das automatisierte Lösen von Anfangswertproblemen mit Hilfe der Datenflussgraphrepräsentation für ODE-Verfahren, ein Framework. Dieses Framework unterstützt nicht nur die Generierung von Datenflussgraphen für die unterstützten Klassen von ODE-Verfahren, sondern auch die Codegenerierung für die Varianten mit Fusion und Tiling sowie die Ausführung des generierten Codes. Dieses Framework ist in der Lage, anhand der Koeffizienten eines RK-Verfahrens, eines PIRK-Verfahrens, eines PEER-Verfahrens oder eines Adams-Bashforth-Verfahrens den dazugehörigen Datenflussgraphen zu erzeugen. Ebenso kann ein Benutzer leicht die Erzeugung eines Datenflussgraphen einer neuen Verfahrensklasse zu dem Framework hinzufügen, wofür unter anderem eine DSL zur Verfügung steht. Zudem kann ein Benutzer das Framework leicht auf neue Zielarchitekturen erweitern.

Unser Framework unterstützt GPUs über CUDA und OpenCL. Im Gegensatz dazu unterstützt unser Framework CPUs über die Laufzeitkompilierung von Kernels zu einer dynamischen Bibliothek. Um automatisch den Code für die Kernels mit Fusion und mit Tiling zu erzeugen, besitzt unser Framework sowohl für GPUs als auch für CPUs optimierte Code-Generatoren. Beide Codegeneratoren beherrschen sämtliche architekturspezifischen Lokalitätsoptimierungen, die wir in dieser Arbeit vorstellen. Dabei erzeugt der GPU-Codegenerator für jedes Kernel Quelltext in CUDA-C oder OpenCL-C, das er dann zur Laufzeit unseres Frameworks zu einem GPU-Kernel kompiliert. Dahingegen erzeugt der CPU-Codegenerator für jedes Kernel ein C-Programm, das er anschließend zur Laufzeit unseres Frameworks zu einer dynamischen Bibliothek kompiliert.

Die so generierten Kernels führt unser Framework dann automatisch in ihrer topologischen Sortierung aus, um darüber die Zeitschrittprozedur des ODE-Verfahrens zu realisieren. Dabei startet unser Framework die GPU-Kernels über CUDA oder OpenCL, während es für die CPU-Kernels die Funktion des Kernels in dessen dynamischer Bibliothek aufruft. Bei dieser automatischen Ausführung beherrscht unser Framework ein automatisches Speichermanagement mit dem Aliasing von Vektoren. Über dieses Aliasing gibt unser Framework einen Vektor, wenn er während der Ausführung nicht mehr benötigt wird, automatisch wieder frei, so dass der für die Ausführung benötigte Speicherplatz reduziert wird. Ebenso implementiert unser Framework eine Unterstützung für NUMA-CPUs, die die Anzahl an Remote-Speicherzugriffen reduziert.

1.4 Gliederung

Diese Arbeit ist in folgende Kapitel untergliedert:

- **2. Verwandte Arbeiten:** Zuerst stellen wir die verwandten Arbeiten vor. Dabei gehen wir vor allem auf Arbeiten zum Autotuning, zur Kernelfusion, zum Tiling und zur Optimierung von ODE-Verfahren ein. Ebenso beschreiben wir, wie sich unsere Arbeit von anderen Arbeiten und von anderen Ansätzen unterscheidet.
- **3. Moderne GPU-Architekturen:** In diesem Kapitel erläutern wir den Aufbau und die Funktionsweise von modernen GPU-Architekturen. Dabei wählen wir eine allgemeine CPU-nahe Abstraktion von modernen GPU-Architekturen, so dass sich ein CPU-affiner Leser leicht in unsere Arbeit einfindet.
- **4. Grundlagen zu ODE-Verfahren:** Hier erläutern wir zunächst die grundlegende Unterteilung von ODE-Verfahren in Einschritt- und Mehrschritt-Verfahren sowie die Unterteilung in implizite und explizite Verfahren. Zudem erklären wir die Konzepte der Ordnung, des lokalen Fehlers und der Schrittweitensteuerung. Zusätzlich stellen wir die wesentlichen von dieser Arbeit abgedeckten ODE-Verfahrensklassen, und zwar die RK-Verfahren, die PIRK-Verfahren, die PEER-Verfahren, die Adams-Bashforth-Verfahren und die Oberklasse der allgemeinen linearen Verfahren vor.
- **5. Klassen von Anfangswertproblemen und Auswahl repräsentativer Testprobleme:** In diesem Kapitel gehen wir zunächst darauf ein, anhand welchen Eigenschaften wir viele wichtige Anfangswertprobleme klassifizieren können. Zusätzlich stellen wir die beiden in dieser Arbeit verwendeten Testprobleme, und zwar *BRUSS2D*, eine zweidimensionale chemische Simulation mit Diffusion, und *NBODY*, eine N-Körper-Simulation, vor.

- **6. Datenflussgraphrepräsentation für ODE-Verfahren:** Hier beschreiben wir unsere Datenflussgraphrepräsentation für ODE-Verfahren, das heißt, wie wir für die von dieser Arbeit abgedeckten ODE-Verfahrensklassen einen Zeitschritt mit Hilfe eines Datenflussgraphen abstrahieren können. Ebenso beschreiben wir, welche Abhängigkeiten in diesem Datenflussgraphen auftreten können und wie wir die Abhängigkeiten über die Zeitschrittgrenzen hinweg modellieren. Zusätzlich stellen wir die Abhängigkeitskette aus RHS $\rightarrow$ LC-Gliedern, die als Grundstruktur in dem Datenflussgraphen von vielen ODE-Verfahren häufig auftritt, vor.
- **7. Grundkonzepte unsere Implementierungen:** Mit diesem sehr kurzen Einschubkapitel führen wir einige sehr wichtige Grundkonzepte, und zwar die abstrakten Kernels, die permanenten Vektoren und die ausführbaren Objekte, ein. Diese Grundkonzepte werden wir in dieser Arbeit über sämtliche Implementierungsvarianten, also über die Basisimplementierung und die Implementierungsvarianten mit Kernelfusion oder Tiling, hinweg verwenden.
- **8. Basisimplementierung:** In diesem Kapitel stellen wir vor, wie wir anhand des Datenflussgraphen von einem ODE-Verfahren automatisch eine Basisimplementierungen für CPUs und GPUs erstellen können. Ebenso führen wir experimentelle Untersuchungen für diese Basisimplementierungen auf einer Test-CPU und Test-GPU durch.
- **9. Kernelfusion:** In diesem Kapitel wenden wir die Kernelfusion an, um die Datenwiederverwendung bei ODE-Verfahren für allgemeine Anfangswertprobleme mit Hilfe von automatisch generierten Implementierungsvarianten zu erhöhen. Dabei stellen wir zunächst Regeln für die Fusion auf, und gehen zusätzlich darauf ein, wie im Detail die Kernelfusion den On-Chip-Speicher der Zielarchitektur zur Einsparung von Speicherzugriffen ausnutzen kann. Ebenso stellen wir die Enabling-Transformationen vor, die uns weitere effiziente Fusionen ermöglichen. Auch definieren wir ein einfaches Performance-Modell, welches über das Datenvolumen die Performance eines Kernels abschätzt, und bewerten mit diesem Performance-Modell mehrere generierte Varianten für das Verner-Verfahren, das DOPRI-Verfahren, die Peer-Verfahren und die PIRK-Verfahren. Zudem beschreiben wir, wie die Kernelfusion auf die Besonderheiten von GPU-Architekturen und CPU-Architekturen eingehen muss, um für beide Architekturen optimierte Implementierungsvarianten mit fusionierten Kernels zu erzeugen. Zuletzt führen wir auf mehreren Test-CPUs und Test-GPUs detaillierte Experimente für die mit der Kernelfusion generierten Implementierungsvarianten durch.
- **10. Tiling:** Dieses Kapitel beschreibt, wie wir bei Anfangswertproblemen mit beschränkter Zugriffsdistanz anhand der Datenflussgraphrepräsentation eines ODE-Verfahrens automatisch Implementierungsvarianten mit Tiling erzeugen können. Dafür stellen wir eine partitionierte Fusion zu einem Kernelpaar vor und geben Regeln für die Anwendung dieser partitionierten Fusion an. Ebenso beschreiben wir, wie sich mit Hilfe der partitionierten Fusion zwei Tiling-Schemata realisieren lassen, und zwar das trapezoidale Tiling-Schema und das hexagonale Tiling-Schema. Zusätzlich realisieren wir das Tiling nicht nur über die Stufen eines ODE-Verfahrens, sondern auch über dessen Zeitschritte. Des Weiteren entwickeln wir für beide Tiling-Schemata einen spezialisierten Autotuning-Ansatz für die Bestimmung der optimalen Tile-Breiten und Tile-Höhen. Auch stellen wir nicht nur das Single-Core-Tiling, sondern auch das Multi-Core-Tiling vor, das heißt, wie mehrere Threads beziehungsweise Kerne an einem Tile zusammenarbeiten können, damit das Tiling für größere Zugriffsdistanzen lohnenswert wird. Des Weiteren beschreiben

wir, wie wir für CPU- und GPU-Architekturen unter der Berücksichtigung der Besonderheiten dieser beiden Architekturen optimierte Implementierungsvarianten mit Tiling erzeugen können. Abschließend führen wir auf mehreren Test-CPUs und Test-GPUs für mehrere expliziten RK-Verfahren eine detaillierte experimentelle Untersuchung der Implementierungsvarianten mit Tiling durch.

- **11. Framework:** In diesem Kapitel stellen wir unser Framework vor, welches wir im Rahmen dieser Arbeit entwickelten und für die Messungen verwendeten. Hierbei gehen wir darauf ein, wie das Framework aufgebaut ist, wie ein Benutzer neue Verfahren in das Framework hinzufügen kann, wie das Framework Code für die Kernels generiert, und wie das Framework ein ODE-Verfahren über dessen Datenflussgraphen und die darinnen enthaltenen Kernels ausführt. Auch stellen wir die DSL vor, mit Hilfe derer ein Benutzer neue ODE-Verfahren zum Framework hinzufügen kann.
- **12. Schlussteil:** In diesem letzten Kapitel geben wir noch einmal eine Zusammenfassung der Arbeit, bewerten, inwieweit wir mit dieser Arbeit unsere gestellten Ziele erfüllen, und geben einen Ausblick, wie sich unsere Arbeit noch erweitern lässt.

1.5 Zugehörige Publikationen

Im Rahmen dieser Dissertation veröffentlichten wir folgende Arbeiten:

- Influence of Locality on the Scalability of Method- and System-Parallel Explicit Peer Methods (2016) [E1]
- Accelerating explicit ODE Methods on GPUs by Kernel Fusion (2018) [E2]
- Exploiting Limited Access Distance for Kernel Fusion Across the Stages of Explicit One-Step Methods on GPUs (2018) [E3]
- Improving Locality of Explicit One-Step Methods on GPUs by Tiling across Stages and Time Steps (2019) [E4]
- Multi-workgroup Tiling to Improve the Locality of Explicit One-Step Methods for ODE Systems with Limited Access Distance on GPUs (2020) [E5]
- Implementation and Optimization of a 1D2V PIC Method for Nonlinear Kinetic Models on GPUs (2020) [E6]
- An In-Depth Introduction of Multi-Workgroup Tiling for Improving the Locality of Explicit One-Step Methods for ODE Systems with Limited Access Distance on GPUs (2021) [E7]

Eine weitere Arbeit zur Generierung von CPU-Implementierungen zum Lösen von Anfangswertproblem über die Datenflussgraphrepräsentation für ODE-Verfahren setzten wir bereits auf, jedoch reichten wir sie noch nicht zur Veröffentlichung ein.

2 Verwandte Arbeiten

2.1 Arbeiten über die Entwicklung neuer ODE-Verfahren

Neben einer Vielzahl von Forschungsarbeiten, die neue ODE-Verfahren mit interessanten numerischen Eigenschaften entwickeln, ohne auf die Aspekte der Informatik einzugehen (siehe zum Beispiel die Arbeiten im Kapitel 4), fallen Forschungsarbeiten, die die Aspekte der Informatik für die Entwicklung neuer, effizienter ODE-Verfahren berücksichtigen, in folgende Kategorien:

- **Entwicklung von Verfahren mit einem hohen Grad an Methodenparallelität:** Hierfür konstruieren Arbeiten ODE-Verfahren, die viele voneinander unabhängige Stufen besitzen. Ein Beispiel hierfür sind die *PIRK-Verfahren* aus den Arbeiten [Z1, Z2], bei welchen ein implizites RK-Verfahren als Basisverfahren verwendet wird, um ein explizites RK-Verfahren mit vielen unabhängigen Stufen zu konstruieren. Ein weiteres Beispiel sind die *Peer-Verfahren* aus der Arbeit [Z3] oder die *parallelen Adams-Bashforth-Verfahren* aus der Arbeit [Z4].
- **Entwicklung von Verfahren mit einem hohen Grad an Zeitparallelität:** Viele moderne Arbeiten versuchen ODE-Verfahren zu konstruieren, die die parallele Berechnung von Zeitschritten erlauben. Eine beliebte Klasse solcher Verfahren sind die sogenannten *Parareal-Verfahren*, die in den Arbeiten [Z5–Z7] behandelt werden.
- **Reduktion des Speicherplatzverbrauchs:** Hierfür versuchen die Arbeiten ODE-Verfahren zu generieren, die zwar eine hohe Ordnung besitzen, aber dafür nur wenige Systemzustände oder zeitliche Ableitungen zeitgleich abspeichern müssen. Hierunter fallen zum Beispiel die *Low-Storage-RK-Verfahren*, welche zum Beispiel von den Arbeiten [Z8, Z9] hergeleitet und untersucht werden.

2.2 Arbeiten im Themenbereich des Autotunings

Das *Autotuning* beschreibt die Fähigkeit eines Programms von selbst, seine Performance, seinen Energieverbrauch oder seinen Speicherplatzverbrauch für ein spezifisches Problem und eine Zielarchitektur beziehungsweise Zielsystem zu optimieren. Ein Überblick über die Thematik des Autotunings ist zum Beispiel in der Arbeit [Z10] zu finden. Dieses Autotuning setzt ein Programm darüber um, indem es selbst Implementierungsvarianten beziehungsweise deren Code generiert, die beste Implementierungsvariante unter einer Menge an vorgegebenen Implementierungsvarianten auswählt, oder indem es die besten Parameter für eine Implementierungsvariante, wie Blockgrößen oder Tile-Größen, ermittelt. Dabei bildet eine Implementierungsvariante mit ihren festgelegten Parametern jeweils eine *Konfiguration* des Programms, wodurch die möglichen Implementierungsvarianten mit ihren möglichen Parametern einen *Konfigurationsraum* aufspannen. Prinzipiell kann ein Programm für sein Autotuning auf diesen Konfigurationsraum eine *vollständige Suche* nach der optimalen Konfiguration durchführen. Jedoch ist in vielen Fällen dieser Konfigurationsraum auf Grund einer kombinatorischen Explosion sehr groß, so dass eine vollständige Suche praktisch nicht durchführbar ist. Jedoch ist für viele Problemstellungen dieser Konfigurationsraum kontinuierlich und zudem verläuft die Performance innerhalb dieses Konfigurationsraums relativ glatt. In diesem Fall kann ein Programm für sein Autotuning, um in diesem Konfigurationsraum zwar nicht die optimale, aber

dennoch eine sehr gute Konfiguration zu finden, Suchalgorithmen wie *genetische Algorithmen*, *Hill-Climbing*, *Nelder-Mead* oder *Simulated Annealing* anwenden. Ein Programm kann, um beim Autotuning die Performance einer Konfiguration zu ermitteln, ein Benchmark mit dieser Konfiguration durchführen oder die Performance dieser Konfiguration mit Hilfe eines *Performance-Modells* oder eines Simulators abschätzen. Mit Hilfe eines Performance-Modells als Heuristik lassen sich zudem oft große Gebiete in dem Konfigurationsraum als sehr suboptimal klassifizieren, wodurch diese von der Suche ausgeschlossen werden können. Generell ist es bei Performance-Modellen problematisch, dass ein allgemeines einfaches Performance-Modell die Performance eines allgemeinen Programms nur unter bestimmten Gegebenheiten zufriedenstellend abschätzen kann. Dahingegen kann ein spezifisches einfaches Performance-Modell die Performance eines spezifischen Programms zwar zufriedenstellend abschätzen, jedoch ist ein solches spezifisches Performance-Modell nicht auf ein allgemeines Programm anwendbar. Um die Performance eines allgemeinen Programms zufriedenstellend abzuschätzen, wird dagegen ein allgemeines komplexes Performance-Modell oder ein Simulator benötigt, wobei der Übergang zwischen den beiden fließend ist. Man unterscheidet zudem zwischen *Offline-Autotuning*, welches während zur Compile-Zeit oder zur Installationszeit eines Programms stattfindet, und *Online-Autotuning*, welches zur Laufzeit eines Programms stattfindet. Im Bereich des Autotunings gibt es unter anderem Arbeiten zu folgenden Themen und Anwendungen:

- **Autotuning für spezifische Probleme:** Viele Arbeiten entwickeln einen Ansatz zum Autotuning für ein spezifisches Problem, und nutzen hierbei die spezifische Berechnungsstruktur des Problems aus, um entweder die für das Autotuning benötigte Zeit zu verkürzen, oder um effizientere Implementierungsvarianten zu generieren. Ein Beispiel hierfür sind die Ansätze für das Autotuning für die *Basic-Linear-Algebra-Subroutinen* (abgekürzt: *BLAS*), insbesondere für die darinnen enthaltene Matrizenmultiplikation. Solche Ansätze werden unter anderem von den Arbeiten [K1] und [K2] entwickelt. Zum Beispiel ermittelt der Ansatz aus der soeben genannten Arbeit [K1] die optimalen Konfigurationen für die BLAS-Subroutinen, welche unter anderem Blockgrößen oder Abrollfaktoren für die Schleifen umfassen, über eine spezialisierte iterative Suche mit einem Performance-Modell, das auf den Blockgrößen und Cache-Größen basiert, als Heuristik. Eine Iteration dieser Suche hält entweder die Blockgrößen oder den Abrollfaktor konstant, um so die Dimensionalität des Suchraums beziehungsweise die Größe des durchsuchten Konfigurationsraums zu reduzieren. Dabei implementierte diese Arbeit zusätzlich einen Codegenerator, welcher den Code für eine BLAS-Subroutine spezifisch für eine gegebene Konfiguration erzeugt. Zusätzlich setzte diese Arbeit ihren Ansatz des Autotunings für die Basic-Linear-Algebra-Subroutinen über eine Bibliothek mit dem Namen ATLAS um. Weitere Beispiele für das problemspezifische Autotuning, sind das Autotuning für die schnelle Fourier-Transformation (siehe zum Beispiel die Arbeit [K3]) oder das Autotuning für Rechenoperationen mit dünnbesetzten Matrizen (siehe zum Beispiel die Arbeit [Z11]). Des Weiteren behandeln Arbeiten zur Kernelfusion auch eine problemspezifische Form des Autotunings, falls sie diese Kernelfusion automatisiert anwenden (siehe Abschnitt 2.3). Zusätzlich lässt sich ein automatisiertes Tiling für Stencil-Codes auch als eine problemspezifische Form des Autotunings betrachten (siehe Abschnitt 2.4). Ebenso existieren Arbeiten, die sich mit dem Autotuning für ODE-Verfahren befassen (siehe Abschnitt 2.5).

- **Autotuning einer Anwendung über generische Frameworks und Bibliotheken:** Mehrere Arbeiten entwickeln generische Frameworks oder Bibliotheken für das Autotuning, mit Hilfe derer ein Benutzer leicht Autotuning in eine beliebige selbstgeschriebene Anwendung integrieren kann. In solchen Frameworks muss ein Benutzer zunächst einen Konfigurationsraum für seine Anwendung definieren. Über diesen Konfigurationsraum führt das Framework eine Suche mit einem durch den Benutzer gewählten Suchalgorithmus durch, wobei das Framework für jede Konfiguration, welche der Suchalgorithmus evaluieren möchte, die Anwendung des Benutzers mit dieser Konfiguration kompiliert und startet. Beispiele für solche Frameworks sind *OpenTuner* aus der Arbeit [K4] und *Actice-Harmony* aus der Arbeit [K5].

- **Suchalgorithmen:** Die Arbeiten zu Suchalgorithmen teilen sich auf in Arbeiten, die für ein allgemeines oder für ein spezifisches Suchproblem einen Suchalgorithmus entwickeln oder bereits vorhandene Suchalgorithmen verbessern, und in Arbeiten, welche sich mit der Übertragung von Suchalgorithmen auf die Problematik des Autotunings beschäftigen. Unter die erstgenannte Gruppe von Arbeiten fallen zum Beispiel die Arbeiten zu Nelder-Mead [Z12], Simulated Annealing (siehe zum Beispiel [Z13] oder [B1]) oder zu den genetischen Algorithmen (siehe zum Beispiel [B2]). Unter die zweitgenannte Gruppe fallen zum Beispiel die Arbeiten [K6] und [K7], welche beide die Geeignetheit von unterschiedlichen Suchalgorithmen für das Autotuning bewerten, indem sie die Laufzeit für das Finden einer guten Konfiguration und die Qualität der besten gefundenen Konfiguration anhand von mehreren Testproblemen, wie zum Beispiel der Matrizenmultiplikation, untersuchen.

- **Performance-Modelle:** Beispiele für spezifische Performance-Modelle sind das soeben genannte Performance-Modell für BLAS-Subroutinen aus der Arbeit [K1] und die in den Abschnitten 2.3 und 2.4 erwähnten Performance-Modelle. Beispiele für allgemeine einfache Performance-Modelle sind das *Roofline-Modell* und das *ECM-Modell*, die wir im Folgenden kurz vorstellen werden. Das *Roofline-Modell* aus der Arbeit [Z14] schätzt die Performance eines Kernels auf einem Prozessor anhand dessen *arithmetischer Intensität*, also wie viel Gleitkommaoperationen das Kernel pro aus dem DRAM geladenen und in den DRAM geschrieben Byte durchführt, und der Peak-Gleitkomma-Performance des Prozessors und der maximalen DRAM-Bandbreite des Prozessors ab. Das Roofline-Modell zeigt zudem das Vorhandensein von eventuellen anderen Flaschenhälsen auf, falls die durch ein Benchmark ermittelte Performance eines Kernels deutlich geringer als die Vorhersage der Performance durch das Roofline-Modell ist. Das *ECM-Modell* (siehe zum Beispiel die Arbeiten [K8] und [K9]) ist eine Erweiterung des Roofline-Modells, welches für seine Performance-Abschätzung zusätzlich die Cache-Bandbreiten, die Cache-Größen und die Workingsets der Speicherzugriffe des Kernels verwendet. Dabei bestimmt es über eine sogenannte Layer-Condition, ob ein Speicherzugriff von einer bestimmten Cache-Ebene abgefangen wird, oder auf den DRAM übergeht. Selbst wenn sowohl das Roofline-Modell als auch das ECM-Modell konzeptuell nicht auf Stencil-Codes beschränkt sind, so werden beide allerdings doch primär dafür verwendet, um die Performance von eben solchen Stencil-Codes abzuschätzen.

2.3 Arbeiten im Themenbereich der Kernelfusion

Die Technik der *Kernelfusion* ist einer der wichtigsten Sate-of-the-Art-Ansätze, um die Lokalität der Speicherzugriffe auf GPUs zu erhöhen. Bei der Technik der Kernelfusion zerlegt man ein ausgewähltes Problem meist in problemspezifische Grundoperationen. Als Nächstes definiert man, wie diese Grundoperationen zu GPU-Kernels zu verschmelzen sind. In einem solchen verschmolzenen GPU-Kernel erfolgt die Datenwiederverwendung meist über Blocking, das heißt, die Berechnungen der verschmolzenen Grundoperationen werden zu nebenläufigen *Blöcken* zusammengefasst und den Workgroups des verschmolzenen GPU-Kernels zugewiesen, wodurch dieses Kernel dann innerhalb eines solchen Blockes die On-Chip-Speicher der GPU (Registersatz, Caches, Scratchpad-Speicher) zur Datenwiederverwendung ausnutzen kann. Eine Automatisierung der Kernelfusion innerhalb eines Programms kann als Ansatz für dessen Autotuning dienen. Im Bereich der Kernelfusion gibt es unter anderem Arbeiten zu folgenden Themen und Anwendungen:

- **Kernelfusion für Datenbankanfragen:** Die Arbeiten [K10, K11] wenden die Kernelfusion an, um Operatoren der relationalen Algebra zu komplexen Datenbankanfragen zu verschmelzen, so dass diese Datenbankanfragen effizient auf GPUs durchgeführt werden können. Ebenso wenden diese Arbeiten die Kernelfission, das Gegenteil von Kernelfusion, an, um komplexere Datenbankanfragen so aufzuspalten, dass wiederum das Kopieren von Daten zwischen GPU und CPU gleichzeitig mit der Ausführung der Datenbankanfragen auf der GPU stattfinden kann.

- **Kernelfusion zur Optimierung der Energieeffizienz:** Die Arbeit [K12] entwickelte eine automatische Technik, um GPU-Kernels zu fusionieren, so dass die Auslastung der unterschiedlichen Ressourcen der GPU erhöht wird, wodurch sich wiederum die Energieeffizienz deutlich steigern lässt.

- **Kernelfusion für Stencils:** Um die effizientesten Fusionen für mehrere Stencils zu finden, schlägt die Arbeit [K13] eine Suche basierend auf einem genetischen Algorithmus vor. Diese Suche verwendet ein Performance-Modell als Suchheuristik, das, ohne Maschinencode zu benötigen, eine obere Grenze für die Performance der möglichen Fusionen der Stencils berechnet. Die Arbeit [K14] automatisiert diese Fusion für Stencils, indem sie einen Source-To-Source-Compiler entwickelt, der intern einen Abhängigkeitsgraphen aufbaut und auf diesen dann die Suchheuristik aus [K13] anwendet.

- **Kernelfusion für Map-, Reduce- und BLAS-Probleme:** Die Arbeit [Z15] schlägt eine automatische Fusion von Kernels für Map-Probleme auf CUDA-GPUs durch einen Source-to-Source-Compiler vor. Dieser Compiler verwendet, um schnell nach effizienten Fusionen zu suchen, einen besonderen Algorithmus, welcher den Suchraum ausästet, und ein Performance-Modell als Suchheuristik, das keinen Maschinencode benötigt. Die Arbeit [Z16] erweitert diesen Compiler zu ineinander verschachtelten Map- und Reduce-Problemen und untersucht die Effizienz dieses Compilers für mehrere ausgewählte BLAS-Operationen. Die Arbeit [K15] verbessert und erweitert die Code-Transformationen dieses Compilers. Zusätzlich fügt diese Arbeit eine Unterstützung von OpenCL zu diesem Compiler hinzu, wodurch der Compiler auch Code für CPUs und anderen Beschleunigern wie FPGAs generieren kann.

- **Kernelfusion für Lösungsalgorithmen von linearen Gleichungssystemen:** Die Arbeit [K16] betrachtet die Kernelfusion anhand des Datenflussgraphen von iterativen Lösungsalgorithmen für lineare dünnbesetzte Gleichungssysteme und wendet diese Überlegungen auf drei *Krylowraum-Löser* mit und ohne Präkonditionierung an. Die Arbeit [Z17] betrachtet die Kernelfusion für iterative Lösungsalgorithmen von linearen Gleichungssystemen mit Pipelining und wendet sie auf das *konjugierte Gradienten-Verfahren* und das *GMRES-Verfahren* an.

- **Kernelfusion mit Multi-Workgroup-Barrieren:** Die Arbeiten [K17, K18] wenden die Kernelfusion mit Multi-Workgroup-Barrieren auf Algorithmen aus der dynamischen Programmierung, Bitonic-Sort und die schnelle Fourier-Transformation an. Dabei wird die Kernelfusion mit Multi-Workgroup-Barrieren aber nur zur Reduktion des Overheads für das Starten von Kernels verwendet, und nicht, um Daten zwischen unterschiedlichen Workgroups wiederzuverwenden oder auszutauschen.

- **Automatische Fusion von Schleifennestern mit anschließendem Offloading auf die GPU:** Die Arbeit [K19] entwickelt ein Compiler-Framework, welches automatisch in dem Quelltext eines gegebenen CPU-Programms Schleifennester erkennt, bei deren Berechnungen ein Auslagern auf die GPU lohnenswert erscheint. Bei diesem Auslagern ist das Compiler-Framework in der Lage, automatisch mehrere Schleifennester, welche auf dieselben Daten zugreifen, zu einem GPU-Kernel zu fusionieren, um so eine Datenwiederverwendung innerhalb eines GPU-Kernels zu erzielen. Ebenfalls ist es in der Lage, ein einziges Schleifennest in mehrere GPU-Kernel aufzuspalten, um so den Workingset der einzelnen Kernel zu verkleinern. Um die Performance der GPU-Kernel zu bewerten beziehungsweise die besten Fusionen und Fissionen zu ermitteln, verwendet dieses Compiler-Framework mehrere Performance-Modelle. Diese Performance-Modelle beinhalten nicht nur eine Abschätzung der Laufzeiten der Kernel, sondern auch den Overhead für die GPU-Initialisierung und den Datentransfer. Letztlich gibt das Compiler-Framework einen OpenMP-Code mit den fusionierten GPU-Kernels aus, welcher dann weiter kompiliert werden kann.

2.4 Arbeiten im Themenbereich der Stencil-Codes und des Tilings

Bei einem *Stencil-Code* handelt es sich um ein Programmfragment, das die zeitliche Entwicklung eines strukturierten Gitters mit meist einer, zwei oder drei Dimensionen über eine Folge von Zeitschritten berechnet. Hierfür iteriert ein Stencil-Code für jeden Zeitschritt einmal über das komplette Gitter, um für jede Gitterzelle des Gitters einen neuen Zustand zu berechnen. Für die Berechnung des neuen Zustands betrachtet der Stencil-Code potentiell nicht nur den aktuellen Zustand der Gitterzelle selbst, sondern auch die aktuellen Zustände der Gitterzellen in deren Nachbarschaft. Diese Nachbarschaft wird im Stencil-Code über einen sogenannten *Stencil* definiert. In einem Stencil-Code werden die Dimensionen des Gitters oft als räumliche Dimensionen bezeichnet, während die Zeitschritte als zeitliche Dimension bezeichnet werden. Aus diesem Grund spannt ein Stencil-Code auf einem 2D-Gitter einen 3D-Iterationsraum auf (x-,y- und Zeit-Dimension), während ein Stencil-Code auf einem 3D-Gitter einen 4D-Iterationsraum aufspannt (x-,y-,z- und Zeit-Dimension). Das *Tiling* ist eine klassische Lokalitätsoptimierungstechnik, die auf CPUs und GPUs verwendet wird, um die Workingsets von einem Iterationsraum, welcher meist über Schleifen definiert ist, zu verkleinern. Hierfür füllt das Tiling den Iterationsraum mit *Tiles* (deutsch: Kacheln) auf, so dass die resultierenden Tiles in die schnel-

leren Ebenen der Speicherhierarchie passen. Dabei erlauben es die Datenabhängigkeiten zwischen den Tiles, dass viele dieser Tiles parallel zueinander berechnet werden können. Während die Parallelisierung der Berechnungen eines einzigen Tiles zwar auch möglich ist, so existieren in einem Tile viele Datenabhängigkeiten, welche wiederum eine häufige Synchronisation erfordern. Tiling ist die wichtigste Optimierungstechnik für Stencil-Codes, da sich darüber die Speicherzugriffslokalität und dadurch wiederum die Performance von Stencil-Codes deutlich erhöhen lässt. Zudem kann eine Automatisierung des Tilings von Stencil-Codes innerhalb eines Programms auch als Ansatz für dessen Autotuning dienen. Da im Themenbereich der Optimierungen für Stencil-Codes und im Themenbereich des Tilings eine unüberschaubare Vielzahl an Arbeiten existiert, stellen wir im folgenden nur einige Arbeiten exemplarisch vor:

- **Polyhedrales Modell:** Ausgehend von einer gegebenen Schleifenstruktur in einem Quelltext, welche arithmetische Ausdrücke mit indexierten Array-Zugriffen beinhaltet, erzeugt das *polyhedrale Modell* eine optimierte Schleifenstruktur oder optimierte Kernel. Ein Beispiel für eine Anwendung dieses polyhedralen Modells ist der PPCG-Compiler aus der Arbeit [Z18]. Die Arbeit [K20] entwickelt basierend auf dem polyhedralen Modell einen allgemeinen Algorithmus, der automatisch ein trapezoidales Tiling für Stencil-Codes auf GPUs erzeugt, und implementiert diesen Algorithmus in den PPCG-Compiler. Die Arbeit [K21] erweitert diesen Compiler um ein Tiling-Schema, das ein hexagonales Tiling entlang der zeitlichen Dimension und einer räumlichen Dimension verwendet, während es entlang den übrigen räumlichen Dimensionen klassische Tilings verwendet. Zusätzlich untersuchen diese beiden Arbeiten beide Tiling-Schemata für mehrere Standard-Stencils. Die Arbeit [T1] entwickelt mehrere Programmtransformationen, welche ein Benutzer auf den abstrakten Syntax-Baum des PPCG-Compilers über eine DSL anwenden kann, um so die Fusion von Schleifen oder ein Tiling für eine Schleife zu spezifizieren. Die Arbeit [K22] stellt einen hierarchischen Online-Autotuning-Ansatz für das polyhedrale Modell vor, und implementiert ihn in den PPCG-Compiler.
- **Multi-Core-Tiling für Stencils:** Das *Multi-Core-Tiling*, das heißt die Zusammenarbeit von mehreren Rechenkernen an einem einzigen Tile, so dass für ein Tile mehr Cache-Speicherplatz zur Verfügung steht, wird zum Beispiel für Stencil-Codes in der Arbeit [Z19] für CPUs ausgenutzt. Diese Arbeit verwendet für ihr Multi-Core-Tiling diamantförmige Tiles, wobei die Abarbeitung innerhalb eines solchen diamantförmigen Tiles diagonal erfolgt. Für die Parallelisierung innerhalb eines Tiles weist die Arbeit jedem der Threads des Tiles in jeder Zeile des Tiles jeweils einen Block von aufeinanderfolgenden Komponenten zu. Nach der Berechnung eines solchen Blockes führen die Threads eine Barriere aus, so dass jeder Thread seinen nächsten Block in der nächsten Zeile berechnen kann. Die Arbeit [K23] erweitert diesen Ansatz auf ein Multi-Core-Tiling mit extrudierten diamantförmigen Tiles. Zusätzlich weist diese Arbeit nun einem Thread in jeder Zeile eines Multi-Core-Tiles die gleichen Komponenten zu, so dass die Datenwiederverwendung über die privaten Caches erfolgen kann. Letztlich testet diese Arbeit das entwickelte Tiling an mehreren 3D-Stencil-Codes zur Simulation von elektromagnetischen Feldern.
- **Locally-Recursive-Non-Locally-Asynchronous-Algorithmen:** Die sogenannten *Locally-Recursive-Non-Locally-Asynchronous-Algorithmen*, welche in der Arbeit [Z20] entwickelt wurden und auch als *LRnLA-Algorithmen* bezeichnet werden, zielen auch auf das Tiling von Stencil-Codes ab. Für dieses Tiling teilen sie den Iterationsraum bestehend aus mehreren räumlichen Dimensionen und einer zeitlichen Dimension rekursiv in polyhedrale Tiles auf. Dabei entspricht die feingranulare rekursive Unterteilung

der lokal-rekursiven Eigenschaft des Algorithmus, die die Datenwiederverwendung über einen cache-oblivious Iterationsraum erhöht. Im Gegensatz dazu entspricht die grobe Unterteilung der Non-Locally-Asynchronous-Eigenschaft des Algorithmus, die eine Menge an Aufgaben darstellt, welche wiederum durch mehrere Threads beziehungsweise mehrere Prozessoren berechnet werden können. Diese Algorithmen wurden von den Arbeiten [K24] und [K25] ausgenutzt, um die Lichtausbreitung in einem Whispering-Gallery-Mode-Micro-Resonator auf CPUs und die Simulation von mehrphasigen Flüssigkeitsströmungen auf GPUs zu simulieren.

- **Domänenspezifische Sprachen und Compiler für Stencils:** Einige Arbeiten entwickeln eine domänenspezifische Sprache für Stencil-Probleme und eventuell zusätzlich für das zu verwendende numerische Lösungsverfahren. Beispiele hierfür sind *PATUS* [K26], *Pochoir* [K27], *Halide* [K28], *PSkel* [W1], *YASK* [W2] und *ExaStencils* [W3]. Aus dieser DSL erzeugen sie dann über mehrere Schritte mit Hilfe eines speziellen Stencil-Compilers einen Programmcode. Dieser Stencil-Compiler wendet bei der Codegenerierung automatische Optimierungen auf unterschiedlichen Abstraktionsebenen an, wie eine automatische Parallelisierung oder ein Tiling.
- **Tiling zur Minimierung des Datentransfers zwischen GPU und CPU:** Die Arbeit [Z21] entwickelt einen automatischen Tiling Mechanismus für Stencil-Codes auf großen Gittern, welche nicht im DRAM der GPU Platz finden, um über das Tiling den Datentransfer zwischen dem DRAM der GPU und dem DRAM der CPU zu minimieren. Somit hat es das Tiling hier zum Ziel, Daten nicht über den On-Chip-Speicher der GPU, sondern über den DRAM der GPU wiederzuverwenden. Diese Arbeit realisiert dies über die Einführung eines virtuellen Speicherraums, wodurch Gittergrößen, die deutlich größer als der DRAM der GPU sind, ermöglicht werden. Schließlich implementiert diese Arbeit ihren Ansatz in das PSkel-Framework.

2.5 Arbeiten im Themenbereich der optimierten Implementierungen für ODE-Verfahren

Während viele Arbeiten existieren, die problemspezifische Optimierungen für eine bestimmte Art von Anfangswertproblemen vorschlagen, so gibt es nur wenige Arbeiten, die die spezifische Struktur von ODE-Verfahren, wie zum Beispiel die Stufenstruktur innerhalb eines Zeitschritts eines ODE-Verfahrens, für optimierte Implementierungen auszunutzen. Innerhalb dieses Themenfelds, das sich mit den optimierten Implementierungen von ODE-Verfahren beschäftigt, gibt es Arbeiten zu folgenden Bereichen:

- **Schleifentransformationen für explizite ODE-Verfahren bei allgemeinen Anfangswertproblemen:** Bei allgemeinen Anfangswertproblemen mit einem beliebigen Speicherzugriffsmuster sind nur Schleifenoptimierungen innerhalb einer Stufe oder zwischen unabhängigen Stufen möglich, aber nicht Schleifenoptimierungen über voneinander abhängigen Stufen oder voneinander abhängigen Zeitschritte. Die Arbeit [Z22] betrachtet Schleifentransformationen, um die Performance eingebetteter RK-Verfahren auf CPUs für solche allgemeinen Anfangswertprobleme zu erhöhen. Diese Schleifentransformationen zielen darauf ab, für die Stufenberechnung die temporale und räumliche Lokalität bei dem Aufsummieren der einzelnen Summanden zu erhöhen. Die Arbeit [Z23] realisiert

und untersucht CPU-Implementierungen von PIRK-Verfahren, die die Methodenparallelität für eine erhöhte Zugriffslokalität ausnutzen, indem sie die unabhängigen Stufen der PIRK-Verfahren über eine verschmolzene Schleifenstruktur berechnen.

- **Tiling für explizite ODE-Verfahren bei Anfangswertproblemen mit beschränkter Zugriffsdistanz:** Bei einer Spezialisierung auf Anfangswertprobleme mit beschränkter Zugriffsdistanz sind zusätzliche Lokalitätsoptimierungen wie beispielsweise ein Tiling über die voneinander abhängigen Stufen oder über die Zeitschritte eines ODE-Verfahrens möglich. Die Arbeit [K29] untersucht für das explizite Euler-Verfahren mehrere Iterationsräume für das hexagonale Tiling über die Zeitschritte auf GPUs. Die Arbeit [K30] betrachtet das Tiling über die Zeitschritte von Adams-Bashforth-Verfahren, während die Arbeit [Z24] das Tiling über die Stufen bei RK-Verfahren betrachtet.

- **Bibliotheken zur Lösung von Anfangswertproblemen:** *ODEINT* [W4] ist eine Header-Only-Opensource-C++-Bibliothek für die Lösung von ODE-Systemen. Es basiert auf Templates und Metaprogramming-Techniken und verfolgt dabei als Ziel, eine hohe Flexibilität und eine gute Performance zu erreichen. ODEINT wird als Teil der weit verbreiteten *Boost-Library* [W5] veröffentlicht, und besitzt eine CUDA-Unterstützung, die wiederum auf der *Thrust* Bibliothek für CUDA [W6] basiert. In unserer Arbeitsgruppe entwickelten wir die Bibliothek E2L (siehe [W7] für einen Auszug hiervon), welche einem Benutzer viele optimierte Implementierungsvarianten von bedeutsamen ODE-Verfahren, wie RK-Verfahren, PIRK-Verfahren oder Adams-Bashforth-Verfahren, zum Lösen von Anfangswertproblemen zur Verfügung stellt.

- **Autotuning für ODE-Verfahren:** Die Arbeit [Z25] entwickelt für PIRK-Verfahren einen Online-Autotuning-Ansatz, das heißt einen Ansatz, welcher das Autotuning zeitgleich zum Lösen des Anfangswertproblems durchführt. Dieser Ansatz wählt innerhalb der ersten wenigen Zeitschritte des PIRK-Verfahrens die beste Implementierungsvariante unter einer Menge von gegebenen Kandidaten und für die Implementierungsvarianten mit Tiling die besten Tile-Größen aus. Vor der Ausführung dünnt dieser Ansatz zusätzlich die Menge der Kandidaten aus, indem er den Synchronisationsoverhead eines jeden Kandidaten abschätzt und diejenigen Kandidaten mit einem sehr großen Synchronisationsoverhead verwirft. Für die Suche nach den besten Tile-Größen verwendet dieser Ansatz eine heuristische empirische Suche, welche durch ein analytisches Performance-Modell geleitet wird. Die Arbeit [K31] entwickelte einen Offline-Autotuning-Ansatz, welcher vor dem Starten des Programms selbst die optimale Implementierungsvarianten anhand eines analytischen Performance-Modells, welches auf dem ECM-Modell basiert, ohne langandauernde Laufzeitmessungen auswählen kann. Dabei generiert das Autotuning in [K31] ausgehend von einer Multi-Level-DSL, in welcher ein ODE-Verfahren durch ein Code-Skeleton und durch Code-Snippets, um dieses Skeleton zu befüllen, beschrieben wird, einen Code für mehrere Varianten, welcher auf die Zielplattform, das ODE-Verfahren und das Anfangswertproblem optimiert ist. Anschließend wendet die Arbeit [K31] ihr Performance-Modell auf die generierten Varianten an, um so eine sehr performante Variante zu bestimmen. Zusätzlich untersucht die Arbeit [K31] ihren Autotuning-Ansatz an mehreren PIRK-Verfahren und für vier unterschiedliche Anfangswertprobleme. Die Arbeit [Z26] baut auf der Codegenerierung für ODE-Verfahren aus der Arbeit [K31] auf, indem sie die Technik des bestärkenden Lernens einsetzt, um aus den per Code-Skeletons und Code-Snippets generierten Implementierungsvarianten eine sehr performante Variante zu bestimmen.

- **Datenflussgraphrepräsentation zum Ausnutzen von Methodenparallelität:** Die Arbeiten [K32] und [Z27] abstrahieren ODE-Verfahren ebenfalls als einen Datenflussgraphen bestehend aus Stufen und deren Abhängigkeiten untereinander. Anschließend partitionierten sie diesen Graphen, um die Methodenparallelität auszunutzen, indem sie die unabhängigen Stufen parallel zueinander von unterschiedlichen Prozessoren berechnen lassen.

2.6 Einordnung unserer folgenden Arbeit

Sämtliche Grundkonzepte, auf die wir in dieser Arbeit zurückgreifen, also im Wesentlichen die dynamische Codegenerierung aus einem Datenflussgraphen, sowie die Lokalitätsoptimierungen der Kernelfusion beziehungsweise des Blockings oder des Tilings sind wohlbekannt. Dementsprechend ist es nicht der Beitrag unserer Arbeit neue Grundkonzepte vorzustellen, sondern wohlbekannte Grundkonzepte für unsere Problemstellung zu modifizieren und geschickt zu kombinieren, sowie die Generierung von optimierten Implementierungen zu automatisieren, um damit das sehr bedeutsame Lösen von Anfangswertproblemen zu beschleunigen. Dementsprechend ordnet sich unsere Arbeit im Vergleich zu anderen relevanten Arbeiten wie folgt ein:

- **Im Vergleich zu anderen Arbeiten zur Kernelfusion:** Während viele vorherige Arbeiten bereits die grundlegende Technik der Kernelfusion behandeln, so spezialisierte sich keine dieser vorherigen Arbeit auf die Kernelfusion für ODE-Verfahren, und versuchte bei dieser Spezialisierung die besondere Struktur von ODE-Verfahren systematisch auszunutzen, und basierend darauf optimierte Implementierungsvarianten sowohl für GPUs als auch für CPUs zu erzeugen.

- **Im Vergleich zu anderen Arbeiten zum Tiling:** Sowohl die Konzepte des trapezoidalen Tiling-Schemas als auch des hexagonalen Tiling-Schemas sind für eindimensionale Stencils sehr gebräuchlich. Unsere Arbeit wendet diese Tiling-Schemata jedoch wieder spezialisiert auf ODE-Verfahren zum Lösen von Anfangswertproblemen mit beschränkter Zugriffsdistanz an und nützt dabei die besondere Stufenstruktur von ODE-Verfahren aus. Im Gegensatz zu unserer Arbeit, die das Tiling für ODE-Verfahren zum Lösen von Anfangswertproblemen mit beschränkter Zugriffsdistanz behandelt, spezialisieren sich LRnLA-Algorithmen auf Stencil-Codes. Folglich nutzen LRnLA-Algorithmen mehrere räumliche Dimensionen für ihr Tiling aus, während der Ansatz in unserer Arbeit nur die Systemdimension des ODE-Systems als einzige räumliche Dimension ausnutzt.

- **Im Vergleich zu anderen Arbeiten zu den optimierten Implementierungen von ODE-Verfahren:** Bisherige Arbeiten zu ODE-Verfahren beschäftigten sich primär damit, zu einem bestimmten gegebenen ODE-Verfahren per Hand eine optimierte Implementierungsvariante zu erstellen. Dahingegen setzen wir in unserer Arbeit auf die automatische Generierung von Implementierungsvarianten aus einer Datenflussgraphrepräsentation für ODE-Verfahren. Auch setzt sich bei bisherigen Arbeiten mit einer Datenflussgraphrepräsentation für ODE-Verfahren der Datenflussgraph nur aus den Stufen des ODE-Verfahrens zusammen, während unsere Datenflussgraphrepräsentation sich aus den einzelnen Grundoperationen des ODE-Verfahrens, also den RHS-Operationen, den LC-Operationen, den RED-Operationen, der Konstantengenerierung und der Schrittweitenkontrolle, zusammensetzt. Dadurch ist unsere Datenflussgraphrepräsentation deutlich

komplexer und umfassender. Ebenso verwenden diese bisherigen Arbeiten die Datenflussgraphrepräsentation nur, um die Methodenparallelität eines ODE-Verfahrens auszunutzen, während unsere Arbeit diese Abstraktion verwendet, um für ein ODE-Verfahren automatisch Implementierungsvarianten mit einer hohen Datenwiederverwendung zu generieren. Des Weiteren beschränkt sich das Autotuning für ODE-Verfahren in den bisherigen Arbeiten primär auf die Suche nach der besten Implementierungsvariante unter einer Menge von gegebenen Kandidaten und auf die Suche nach den besten Performance-Parametern, wie die Blockgrößen oder den Abroll-Faktoren für die Schleifen. Selbst obwohl die automatische Codegenerierung für ODE-Verfahren bereits in anderen Arbeiten mit Hilfe von Code-Skeletons implementiert wurde, so ist diese Codegenerierung deutlich weniger flexibel als die Codegenerierung aus einem Datenflussgraphen, da hier die Code-Skeletons und die Code-Snippets für ein ODE-Verfahren vom Benutzer definiert werden müssen. Allerdings ließe sich dieser Ansatz der Code-Skeletons und der Codegenerierung aus einem Datenflussgraphen miteinander kombinieren, so dass der Codegenerator aus einem Datenflussgraphen beziehungsweise aus den im Datenflussgraphen enthaltenen Kernels über Code-Skeletons und Code-Snippets eine Vielzahl an Varianten generiert, in welchen man dann über ein Performance-Modell die beste Variante bestimmen könnte.

3 Moderne GPU-Architekturen

3.1 Allgemeines

GPUs sind Prozessoren, die auf großen Rechendurchsatz für das Lösen von massiv datenparallelen Problemen via *Single Instruction, Multiple Data* (*SIMD*) ausgelegt sind. Aus diesem Grund besitzen GPUs, verglichen mit CPUs, ein breites Speicherinterface, kleine Caches und viele Kerne mit breiten Vektoreinheiten, vielen parallelen Threads, und einem großen Registersatz. GPUs sind zudem weitestgehend *Load-Store-Architekturen*, das heißt alle Operanden von arithmetischen Instruktionen müssen sich in den Registern befinden und Zugriffe auf den Speicher müssen explizit mit dedizierten Lade- und Schreibinstruktionen erfolgen.

Aktuell gibt es drei größere Hersteller von GPUs im Desktop- und im Server-Bereich: NVIDIA, AMD und Intel. Obwohl es kleinere und größere Unterschiede sowohl zwischen den GPU-Architekturen der drei Hersteller als auch zwischen den GPU-Generationen innerhalb eines Herstellers gibt, so sind der grundlegende Aufbau und die Funktionalitäten von modernen GPU-Architekturen sehr ähnlich. Deshalb versuchen wir im Folgenden allgemein den Aufbau und die Funktionalitäten einer modernen GPU-Architektur zu erläutern. Der konkrete Aufbau von diversen modernen GPU-Architekturen wird im Abschnitt 13.5 gezeigt.

Dennoch können konkrete GPU-Architekturen von dieser folgenden allgemeinen Darstellung abweichen. Gerade bei älteren GPU-Architekturen sind die Abweichungen im Aufbau größer und viele der hier vorgestellten Funktionalitäten nicht oder nur eingeschränkt vorhanden. Auch kann bei einer konkreten GPU-Architektur eine zusätzliche Cache-Ebene existieren sowie eine Cache-Ebene anders zwischen den Kernen der GPU geteilt oder eine Cache-Ebene nicht vorhanden sein. Dadurch verschieben sich dann die anderen Cache-Ebenen im Vergleich zu den folgenden Erläuterungen entsprechend. Auch vergrößerten NVIDIA und AMD in 2020 die Caches auf ihren zwei neuen GPU-Architekturen, der (Compute-)Ampere-Architektur und RDNA-2.0-Architektur, verglichen mit ihren Vorgängern, deutlich. Zudem fügte AMD bei der RDNA-2.0-Architektur zusätzliche Ebenen zu der Cache-Hierarchie hinzu (siehe Abschnitt 13.5). Wenn sich diese Entwicklung fortsetzt, dann werden GPUs in naher Zukunft standardmäßig ähnlich große Caches und komplexe Cache-Hierarchien wie CPUs besitzen.

Ebenso implementieren GPUs viele Funktionalitäten, die fast ausschließlich für die Computergraphik verwendet werden können, wie eine Hardware-Beschleunigung für die Rastergraphik, für das Ray-Tracing und für das Sampling von Texturen. Letztlich besitzen GPU-Architekturen auch oft zusätzliche spezialisierte Textur-Caches, die nicht kohärent gehalten werden. Dadurch kann sie ein Kernel auch nur für Daten verwenden, die sich während dessen Laufzeit nicht verändern. All diese Graphik-Funktionalitäten inklusive der Textur-Caches werden wir in den folgenden Erläuterungen vernachlässigen.

Das folgende Kapitel stützt sich auf viele Whitepapers über GPU-Architekturen, Programming Guides und ISA-Dokumentationen von NVIDIA [T2–T9], AMD [T10–T15] und Intel [T16–T19]. Allerdings wählen all diese Quellen, um den Aufbau und die Funktionsweise von ihrer jeweiligen GPU-Architektur zu beschreiben, sehr uneinheitliche Terminologien und Abstraktionen, welche zusätzlich widersprüchlich zu den gängigen Konventionen der Informatik sind. Deshalb passten wir für folgendes Kapitel die Terminologien und Abstraktionen aus all diesen Quellen an, so dass sie möglichst CPU-nah sind. Aus diesem Grund stellt dieses Kapitel keine reine Zusammenfassung, sondern eine deutliche Überarbeitung und Homogenisierung dieser Quellen dar, und ist in dieser Art einzigartig. Falls der Leser an weiterführender Literatur inter-

essiert ist, die ähnliche Abstraktionen wie diese Arbeit verwendet, so empfehlen wir ihm eine aktuelle Ausgabe des Buchs Computer Architecture: A Quantitative Approach [B3] oder die GPU-Dokumentationen von Intel [T16–T19], da diese ebenfalls noch sehr CPU-nah sind.

3.2 Terminologie

Bedauerlicherweise existiert keine Terminologie für die Hardware-Architekturen von GPUs, die allgemein akzeptiert ist, sondern jeder GPU-Hersteller (NVIDIA, AMD, Intel) und jede API zum Programmieren von GPUs (zum Beispiel CUDA, OpenCL, DirectX, OneAPI) verwendet eine eigene GPU-Terminologie.

Erschwerend kommt hinzu, dass NVIDIA und AMD die Begrifflichkeiten in ihren Terminologien oft entgegen gängiger Konventionen der Informatik wählen. Insbesondere gilt dies für den Begriff des Threads und des Kerns beziehungsweise des Cores. So bezeichnen NVIDIA und AMD eine *SIMD-Lane* eines (Hardware-)Threads als Thread und einen (Hardware-)Thread als *Warp* beziehungsweise *Wavefront*. Ebenso bezeichnet NVIDIA eine einzige Lane einer SIMD-FPU als einen *CUDA-Core*, und AMD bezeichnet eine solche Lane als *Stream-Core*. Dies widerspricht jedoch der geläufigen Definition eines Kerns (siehe zum Beispiel [B4] oder [B5]) als eine weitgehend autonome Recheneinheit mit einem eigenen Steuerwerk, Rechenwerk, Registersatz und eventuellen privaten Caches. Somit besitzen moderne GPUs unter Verwendung dieser GPU-Terminologien verglichen mit modernen CPUs unter der Verwendung der gemein gebräuchlichen CPU-Terminologie sehr viele Kerne und können sehr viele Threads parallel ausführen. Zusätzlich bezeichnen NVIDIA und AMD ihre GPU-Architekturen als skalare Architekturen, obwohl es sich bei ihnen, zumindest gemäß der gemein gebräuchlichen Definition, um SIMD-Architekturen handelt. So existieren in der GPU-Terminologie von NVIDIA auch keine gängigen SIMD-Begriffe wie Vektoreinheiten, Vektorregister, Scatter- und Gather-Instruktionen. Die GPU-Terminologie von NVIDIA bezeichnet zwar sehr spezielle Typen von Instruktionen als SIMD-Instruktionen. Aber auch diese Bezeichnung verstößt gegen die gängigen Konventionen, da prinzipiell fast ausschließlich SIMD-Instruktionen auf NVIDIA-GPUs existieren.

Um in dieser Arbeit solche Missverständnisse auszuräumen und um dem CPU-affinen Leser den Einstieg zu erleichtern, verwenden wir in dieser Arbeit deshalb eine selbst erschaffene Terminologie, die die Begriffe im Einklang zur gemein gebräuchlichen CPU-Terminologie, wie sie zum Beispiel von den Büchern [B4], [B5] oder [B3] verwendet wird, definiert. Ein vergleichender Überblick zwischen den wichtigsten Begriffen unserer Terminologie und den Terminologien von NVIDIA, AMD, und Intel befindet sich in Tabelle 3.1.

Wir sind uns darüber im Klaren, dass bei vielen konkurrierenden, und teils widersprüchlichen Terminologien eine weitere neue zusätzliche Terminologie die bereits bestehende Verwirrung weiter und unnötig vergrößern kann. Jedoch vermuten wir, dass der Leser sich leicht in unsere Terminologie einfinden kann, da sie sehr CPU-nah ist, die Begriffe gemäß ihrer ursprünglichen Definition verwendet, und bewusst unnötige Neologismen vermeidet. Wenn wir dennoch einen Neologismus verwenden, versuchen wir ihn so zu wählen, dass der Leser intuitiv die Bedeutung hinter einem Begriff versteht. Zusätzlich ist unsere Terminologie in unseren Augen, da sie die Begriffe so verwendet, wie sie ursprünglich definiert wurden, wissenschaftlich exakter. Diese Vorteile unserer Terminologie werden in unseren Augen bereits sehr deutlich beim Lesen von der Tabelle 3.1 gezeigt.

Unsere Terminologie	NVIDIA	AMD	Intel
Kern	Streaming-Multiprocessor	Compute-Unit, Dual-Compute-Unit	Subslice
Subkern	Processing-Block, Sub-Core	SIMD	Execution-Unit
Workgroup	Threadblock	Workgroup	Workgroup
Workitem	Thread	Workitem	Kernel-Instance
Thread	Warp	Wavefront	Thread
SIMD-Lane eines Threads	Thread (meistens), Lane (seltenst)	Workitem in a Wavefront, Thread	SIMD-Lane
Vektor-ALU+FPU	?	Vector-ALU	SIMD-FPU, SIMD-ALU
SIMD-Lane in einer Vektor-ALU+FPU	CUDA-Core	Stream-Processor, Stream-Core, ALU-Unit	SIMD-Lane
Heap-Speicher	≈ Global-Memory	≈ Global-Memory	≈ Global Memory
Stack-Speicher	Local-Memory	Scratch-Memory, Private-Memory	?
Scratchpad-Speicher	Shared-Memory	Local-Data-Share	Shared-Local-Memory
Skalarer Ausführungspfad	Uniform-Datapath	Scalar-Unit usw.	n/a

Tabelle 3.1: Zentrale Begriffe der GPU-Terminologien unterschiedlicher Hersteller.

3.3 SIMD

Eine *Vektor-Architektur* oder *Single-Instruction-Multiple-Data-Architektur* (*SIMD-Architektur*) führt Instruktionen auf einem Vektor von Datenelementen aus. Dies steht im Gegensatz zu einer sogenannten *skalaren Architektur* oder *Single-Instruction-Single-Data-Architektur* (*SISD-Architektur*), die jede Instruktion nur auf einem einzigen Datenelement ausführt. Da eine SIMD-Architektur für die gleiche Peak-Performance weniger Kontrolllogik als eine SIMD-Architektur mit geringerer Vektorbreite oder eine SISD-Architektur benötigt, ist sie auch kostengünstiger in der Fertigung und energieeffizienter. Jedoch verliert eine SIMD-Architektur auch viel ihrer Rechenleistung, wenn die Berechnungen nur wenig datenparallel oder auch nur schlecht vektorisiert sind.

Bei SIMD-Architekturen bezeichnet man die Anzahl der Komponenten in einem Vektor als

SIMD-Breite oder als *Vektorbreite*. Alternativ kann man auch die Vektorbreite als die Anzahl der Bits in einem Vektor definieren. In folgender Arbeit definieren wir immer, sofern nicht explizit erwähnt, die SIMD-Breite über die Anzahl der Komponenten in einem Vektor. Man unterscheidet hier noch einmal zwischen der Anzahl der Komponenten pro Instruktion (*logische SIMD-Breite* oder *logische Vektorbreite*), der Anzahl derjenigen Komponenten, die ein Vektorrechenwerk gleichzeitig verarbeiten kann (*physikalische SIMD-Breite* oder *physikalische Vektorbreite*) und wie viele Kontrollflüsse ein einzelner Thread per SIMD-Instruktionen sowie mit der Hilfe von Predication oder zusätzlicher Kontrolllogik abarbeiten kann. Dies bezeichnen wir im Folgenden als *Kontrollvektorbreite*. Man bezeichnet einen solchen Kontrollfluss innerhalb eines Threads oft als eine *SIMD-Lane* dieses Threads.

Auf GPU-Architekturen ist die logische Vektorbreite für 4-Byte-Datentypen und 8-Byte-Datentypen identisch und gleich der Kontrollvektorbreite. Für 1-Byte-Datentypen und 2-Byte-Datentypen unterscheidet sich der Zusammenhang zwischen logischer Vektorbreite und Kontrollvektorbreite je nach GPU-Architektur. Auf einem Teil der GPU-Architekturen ist die logische Vektorbreite für 1-Byte-Datentypen und 2-Byte-Datentypen ebenfalls gleich der Kontrollvektorbreite. Es existieren jedoch auch GPU-Architekturen, auf welchen die logische Vektorbreite von 1-Byte-Datentypen und 2-Byte-Datentypen viermal so groß ist, wie die Kontrollvektorbreite, wodurch sich jeweils bei 1-Byte-Datentypen vier Komponenten und bei 2-Byte-Datentypen zwei Komponenten einen Kontrollfluss teilen.

Auf den GPU-Architekturen von Intel und auf den RDNA-GPU-Architekturen von AMD ist die Kontrollvektorbreite und damit die logische Vektorbreite konfigurierbar. So beherrscht zum Beispiel die RDNA-Architektur einen Modus, bei welchem die Kontrollvektorbreite und die logische Vektorbreite für 4-Byte-Datentypen und 8-Byte-Datentypen 32 Komponenten groß sind, während in einem weiteren Modus die Kontrollvektorbreite und die logische Vektorbreite für 4-Byte-Datentypen und 8-Byte-Datentypen 64 Komponenten groß sind.

3.4 Aufbau

Eine moderne GPU-Architektur (siehe Abbildung 3.1 für ein Blockdiagramm einer solchen Architektur) besitzt beziehungsweise besteht in der Regel aus folgenden Bestandteilen:

- **DRAM:** GPU-Architekturen verwenden meist DRAM-Technologien, die auf eine große Bandbreite optimiert worden sind (zum Beispiel GDDR6 oder HBM). Analog, wie auf CPUs, dient der DRAM als Arbeitsspeicher, um die Instruktionen der Programme beziehungsweise der Kernels, die Konstanten, den Heap und die Stacks der Threads abzuspeichern.

- **Speicherinterface:** Das Speicherinterface auf GPUs besitzt viele Kanäle und ist dadurch sehr breit. Jeder dieser Kanäle besitzt seinen eigenen DRAM-Controller. Dadurch erreichen GPUs in Kombination mit dem hochbandbreitigen DRAM eine vergleichsweise große Speicherbandbreite.

- **L2-Cache:** Der L2-Cache speichert Zugriffe auf den DRAM zwischen und ist zwischen allen GPU-Kernen geteilt. Verglichen mit dem geteilten L3-Caches auf aktuellen Desktop- oder Server-CPUs ist er jedoch sehr klein. Er ist zudem in mehrere parallele Partitionen unterteilt und befolgt bei Schreibvorgängen eine *Write-Allocate-Policy* mit *Write-Back* und *No-Fetch-on-Write*. Die No-Fetch-on-Write-Policy besagt hierbei, dass

bei einem Write-Miss die Cache-Line zwar alloziert, nicht aber aus dem DRAM geladen wird. Ein Laden aus dem DRAM erfolgt erst, falls ein Kern auf diese Cache-Line lesend zugreift. Ebenso wird bei dem Write-Back einer Cache-Line, die per No-Fetch-on-Write alloziert wurde, über eine Write-Mask sichergestellt, dass nur die modifizierten Bytes in den DRAM zurückgeschrieben werden. Folglich verhindert eine Write-Allocate-Policy mit No-Fetch-on-Write denjenigen Overhead an DRAM-Transfers, der bei einer Write-Allocate-Policy mit Fetch-on-Write auftritt (siehe zum Beispiel [Z28] für eine ausführliche Erörterung von dem Fetch-on-Write- und dem No-Fetch-On-Write-Verhalten). Des Weiteren besitzt der L2-Cache sogenannte atomare Einheiten, die diverse atomare Operationen, in der Regel Compare-And-Swap, Integerminimum, Integermaximum, Integeraddition und die Gleitkommaaddition, direkt auf den Daten im L2-Cache ohne ein Hin-und-Zurück über die Kerne der GPU ausführen können.

- **Verbindungsnetzwerk:** Dieses verbindet die Kerne mit den Partitionen des L2-Caches und ist auf modernen GPUs als ein Crossbar-Netzwerk realisiert.
- **Task-Scheduler:** Der *Task-Scheduler* arbeitet mehrere *Task-Queues* ab. Bei den Tasks in diesen Task-Queues kann es sich entweder um auszuführende Kopieroperationen oder um auszuführende Kernels handeln. Bei der Abarbeitung der Task-Queues berücksichtigt der Task-Scheduler diejenigen Abhängigkeiten zwischen den Tasks, die die Host-Anwendung zuvor über *Events* definiert hat. Zusätzlich weist der Task-Scheduler während der Kernelausführung die Workgroups den Kernen der GPUs zu.
- **Copy-Engines:** Die *Copy-Engines* kopieren Daten zwischen dem DRAM der CPU und dem DRAM der GPU oder auch nur Daten innerhalb des DRAMs der GPU.
- **GPU-Kerne:** Die *GPU-Kerne* führen analog zu CPU-Kernen die Programme beziehungsweise die Kernels mit ihren Berechnungen aus.

Jeder GPU-Kern (siehe Abbildung 3.2 für ein beispielhaftes Blockdiagramm) lässt sich wiederum wie folgt feiner gliedern:

- **Scratchpad-Speicher:** Dieser kleine Scratchpad-Speicher besitzt eine hohe Bandbreite und ist in viele Bänke unterteilt. Diese Bänke können unabhängig voneinander adressiert werden, wodurch der Scratchpad-Speicher eine feine Zugriffsgranularität besitzt. Über den Scratchpad-Speicher können sich die Workitems beziehungsweise die Threads einer Workgroup untereinander Daten teilen. Zusätzlich sind analog wie beim L2-Cache atomare Einheiten vorhanden, welche atomare Operationen direkt auf dem Scratchpad-Speicher ohne ein Hin-und-Zurück über die Rechenwerke des Kerns ausführen.
- **L1-Instruktionscache:** Dieser speichert die Instruktionen des auszuführenden Kernels zwischen.
- **L1-Datencache:** Der L1-Datencache ist verglichen mit den privaten L1- und L2-Caches von modernen CPUs sehr klein und speichert die Zugriffe eines Kernels auf den DRAM der GPU zwischen. Bei Schreibvorgängen befolgt er eine Write-Allocate-Policy mit Write-Through.
- **Subkerne:** Jeder GPU-Kern ist noch einmal in mehrere *Subkerne* gegliedert, wobei jeder Subkern im Wesentlichen aus einem Thread-Scheduler, einem Vektorregistersatz,

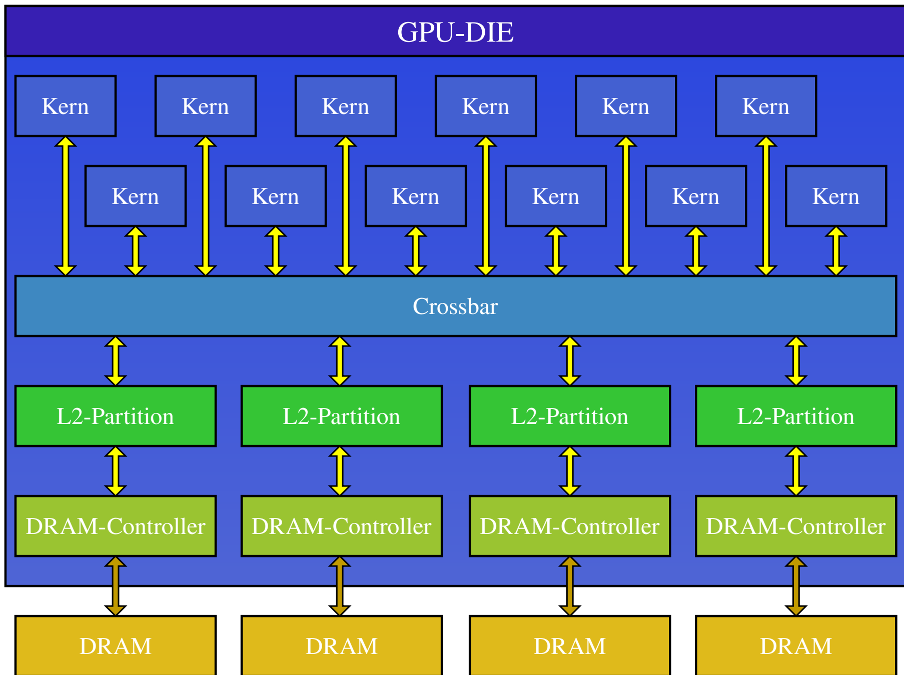


Abbildung 3.1: Schematisches Blockdiagramm einer modernen GPU. Eine konkrete moderne GPU besteht aus deutlich mehr Rechenkernen, L2-Partitionen und Speicherkanälen, als in diesem schematischen Beispiel gezeigt werden.

einer Vektor-Load-Store-Einheit und aus Vektorrechenwerken besteht. Je nach GPU-Architektur können jedoch manche dieser Vektorrechenwerke, die Vektor-Load-Store-Einheit und auch Teile der Thread-Scheduling-Logik zwischen den Subkernen geteilt sein.

Dementsprechend besteht ein Subkern aus folgenden Einheiten:

- **Thread-Scheduler:** Der *Thread-Scheduler* führt viele Threads gleichzeitig per *Fine Grained Interleaved Multi-Threading* aus. Bei diesem Fine Grained Interleaved Multi-Threading wählt der Thread-Scheduler gemäß einem Score-Board jeden Takt meist einen anderen Thread aus, der bereit ist seine nächste Instruktion auszuführen, und gibt diese nächste Instruktion zur Abarbeitung an die Rechenwerke heraus. Dadurch arbeitet ein GPU-Kern die Instruktionen eines einzelnen Threads gemäß einer einfachen In-Order-Pipeline ab.

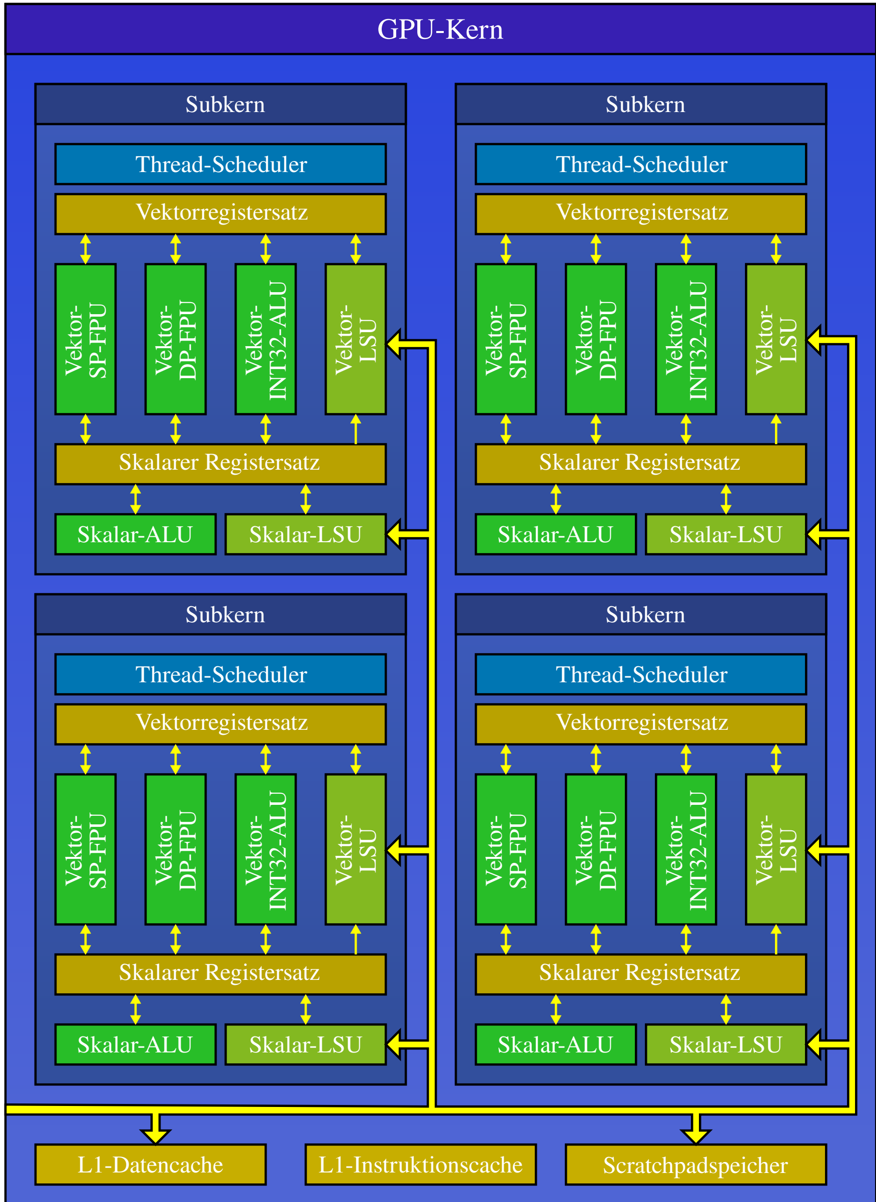


Abbildung 3.2: Schematisches Blockdiagramm eines Kerns einer modernen GPU. Der Kern einer konkreten modernen GPU-Architektur kann eine andere Anzahl an Subkernen und andere Vektorrechenwerke als der Kern in diesem Beispiel besitzen.

- **Ausführungspfad für Vektorinstruktionen:**
  - **Vektorregistersatz:** Der Vektorregistersatz ist verglichen mit CPUs sehr groß. Zudem ist die Anzahl an Registern pro Thread nicht fest, sondern ein Kernel kann für sich eine bestimmte Anzahl an Registern pro Thread allozieren. Dadurch verringert sich aber auch die Anzahl der von einem Kern parallel ausgeführten Threads, sofern ein bestimmter Schwellwert an allozierten Registern pro Thread überschritten wird. Des Weiteren ist jede der Lanes in einem Vektorregister 4 Byte breit. Deshalb kann eine jede dieser Lanes einen 4-Byte-Datentypen abspeichern. Um einen 8-Byte-Datentypen abzuspeichern, werden zwei Vektorregister benötigt, wobei jede der Lanes die niederen 4 Byte des 8-Byte-Datentyps jeweils im ersten Vektorregister und die höheren 4 Byte jeweils im zweiten Vektorregister abspeichert. Wie Datentypen unterschiedlicher Breite in dem Vektorregistersatz abgelegt werden, wird von Abbildung 3.3 gezeigt.
  - **Vektorrechenwerke:** Ein jeder Subkern besitzt mehrere Vektorrechenwerke, wobei jedes Vektorrechenwerk ähnlich wie auf einer CPU auf einen oder mehrere Typen von Operationen spezialisiert ist. Geläufig sind folgende Spezialisierungen:
    - **Vektor-SP-FPU:** Gleitkommaoperationen einfacher Genauigkeit
    - **Vektor-DP-FPU:** Gleitkommaoperationen doppelter Genauigkeit
    - **Vektor-INT32-ALU:** 32-Bit-Integeroperationen und logische Operationen
    - **Vektor-SP-SFU:** Sonderfunktionen (Kehrwert, Winkelfunktionen, Exponentialfunktion und Logarithmus)
    - **Vektor-SP-FPU+INT32-ALU:** Kombination von SP-FPU und INT32-ALU
    - **Vektor-SP-FPU+DP-FPU+INT32-ALU:** Kombination von SP-FPU, DP-FPU und INT32-ALU
    - **Tensor-HP-FPU:** Tensoroperationen mit halber Genauigkeit

    Diese Vektorrechenwerke können unterschiedliche physikalische Vektorbreiten besitzen, so dass wichtige Operationen einen höheren Durchsatz erhalten als weniger wichtige Operationen. Außerdem beherrschen die SP-FPUs und DP-FPUs moderner GPUs FMA-Instruktionen, wodurch jede Lane einer SP-FPU beziehungsweise DP-FPU pro Takt 2 FLOPs durchführen kann.
  - **Vektor-Load-Store-Einheit:** Die Vektor-Load-Store-Einheit arbeitet die Speicherzugriffe eines Kernels, insbesondere die *Gather-Instruktionen* und *Scatter-Instruktionen*, auf den Scratchpad-Speicher und den Arbeitsspeicher, ab. Zusätzlich führt sie bei Gather- und Scatter-Instruktionen auf den Arbeitsspeicher das *Coalescing* durch.
- **Ausführungspfad für skalare Instruktionen:** Dieser skalare Ausführungspfad besteht aus einem skalaren Registersatz, einem skalaren Rechenwerk, einer skalaren Load-Store-Einheit und einem skalaren Cache. Über diesen skalaren Ausführungspfad können GPU-Architekturen effizient *Predication* und ihren Kontrollfluss implementieren. Zudem können sie mit Hilfe ihres skalaren Ausführungspfads auch viele Loop-Header und viel Index-Arithmetik effizienter berechnen. Auch können die skalaren Register als Operand für spezielle Skalar-Vektor-Instruktionen dienen, mit denen die GPU die sehr häufig in Kernels vorkommenden Skalar-Vektor-Operationen effizient berechnen kann. Dieser skalare Ausführungspfad ist nur auf AMD-GPUs seit der GCN-Architektur und auf NVIDIA-GPUs seit der Turing-Architektur vorhanden.

1-Byte-Datentyp mit einer Komponente pro SIMD-Lane

	1. Lane	2. Lane	3. Lane	4. Lane
Vektorregister	1 B	1 B	1 B	1 B

1-Byte-Datentyp mit vier Komponenten pro SIMD-Lane

	1. Lane	2. Lane	3. Lane	4. Lane
Vektorregister	1 B 1 B 1 B 1 B	1 B 1 B 1 B 1 B	1 B 1 B 1 B 1 B	1 B 1 B 1 B 1 B

2-Byte-Datentyp mit einer Komponente pro SIMD-Lane

	1. Lane	2. Lane	3. Lane	4. Lane
Vektorregister	2 B	2 B	2 B	2 B

2-Byte-Datentyp mit zwei Komponenten pro SIMD-Lane

	1. Lane	2. Lane	3. Lane	4. Lane
Vektorregister	2 B 2 B	2 B 2 B	2 B 2 B	2 B 2 B

4-Byte-Datentyp

	1. Lane	2. Lane	3. Lane	4. Lane
Vektorregister	4 B	4 B	4 B	4 B

8-Byte-Datentyp

	1. Lane	2. Lane	3. Lane	4. Lane
1. Vektorregister 2. Vektorregister	8 B	8 B	8 B	8 B

Abbildung 3.3: Anordnung von Datentypen unterschiedlicher Breite in dem Vektorregistersatz einer GPU. Für diese Abbildung nehmen wir an, dass ein Vektorregister auf einer beispielhaften GPU vier SIMD-Lanes besitzt, wobei jede SIMD-Lane eines Vektorregisters vier Byte abspeichern kann. Dadurch kann ein Vektorregister 16 Vektorkomponenten für einen 1-Byte-Datentypen, 8 Vektorkomponenten für einen 2-Byte-Datentypen und 4 Vektorkomponenten für einen 4-Byte-Datentypen abspeichern. Zudem benötigt die GPU zwei Vektorregister, um 8 Vektorkomponenten für einen 8-Byte-Datentypen abzuspeichern, wobei jede Vektorkomponente des 8-Byte-Datentyps über zwei Vektorregister verteilt vorliegt. Es gilt zudem, dass auf vielen GPUs bei 1-Byte-Datentypen und 2-Byte-Datentypen die Vektorrechenwerke pro SIMD-Lane nur einen Wert eines solchen Datentyps verarbeiten können, wodurch die übrigen Bytes von jeder SIMD-Lane im Vektorregister unbenutzt bleiben.

Letztlich gilt anzumerken, dass die Wahl des Begriffes des GPU-Kerns und des Subkerns in unserer Terminologie ebenfalls problematisch ist. Denn je nachdem, wie viel Kontrolllogik oder wie viele Rechenwerke zwischen den Subkernen eines GPU-Kerns bei einer konkreten GPU-Architektur geteilt sind, und wie viel Autonomie man von einem GPU-Kern erwartet, könnte man argumentieren, dass ein Subkern bereits ein eigenständiger GPU-Kern sei. Dementsprechend müsste man einen anderen Begriff für das, was wir in dieser Arbeit als GPU-Kern bezeichnen, verwenden. Da ein sehr ähnliches Definitionsproblem bei dem Begriff des CPU-Kerns bei den AMD-CPUs mit der Bulldozer-Architektur bereits einen langen Rechtsstreit vor einem amerikanischen Gericht verursacht hat (siehe zum Beispiel [W8]), werden wir in dieser Arbeit nicht weiter darüber diskutieren.

Im Folgenden werden wir kurz noch auf drei interessante besondere Bestandteile beziehungsweise Funktionalitäten, die bei mehreren, aber nicht bei allen modernen GPU-Architekturen vorhanden sind, eingehen:

- **L1-Datencache und Scratchpad-Speicher im selben physikalischen Speicher:** Auf GPU-Architekturen von NVIDIA teilt sich der L1-Datencache oft mit dem Scratchpad-Speicher denselben physikalischen Speicher, wobei die Aufteilung dieses physikalischen Speichers zwischen dem Scratchpad-Speicher und dem L1-Datencache konfigurierbar ist.

- **Last-Level-Cache:** Dieser zusätzliche Last-Level-Cache ist bei den wenigen Architekturen, die ihn implementieren, sehr viel größer als der L2-Cache, und ähnlich groß wie ein L3-Cache auf modernen CPUs. Da es nur wenige GPU-Architekturen mit einem solchen Last-Level-Cache gibt (diverse IGP-Architekturen von Intel und die RDNA2-Architektur von AMD), ist es nur schwer möglich, die zukünftige Entwicklung dieses Caches abzuschätzen. Es fällt zudem schwer, diesem Last-Level-Cache eine vernünftige Nummer zuzuweisen, denn bei den wenigen Architekturen mit diesem Last-Level-Cache weicht die übrige Cache-Hierarchie von den Erläuterungen in dieser Arbeit ab.

- **Konstantenspeicher:** Auf GPU-Architekturen von NVIDIA existiert zusätzlich ein sogenannter *Konstantenspeicher* mit einem eigenen Speicherraum, welcher von einem Kernel heraus nur gelesen werden kann. Die Adressen im Speicherraum des Konstantenspeichers können, wie bei einer *Register-Memory-Architektur*, direkt als Operand für eine SIMD-Instruktion dienen, wobei die Daten an diesen Adressen dann automatisch während der Ausführung der Operation aus dem Konstantenspeicher geladen werden. Durch einen besonderen Konstanten-Cache ist das Lesen aus dem Konstantenspeicher bei einem Cache-Treffer genauso schnell wie das Lesen aus einem Register. Dadurch können GPUs mit diesem Konstantenspeicher effizient Skalar-Vektor-Instruktionen implementieren.

3.5 Kernelausführung

GPUs führen Programme aus, die allgemein als *Kernel* bezeichnet werden. Ein Kernel wird über eine *Kernelfunktion*, ähnlich einer Einstiegsfunktion eines CPU-Threads, definiert. Diese wird in der Regel in einer C-ähnlichen Sprache geschrieben, und dann mit einem GPU-Compiler speziell für die Architektur der Ziel-GPU kompiliert.

Um ein solches kompiliertes Kernel auf einer GPU zu starten, muss die Anwendung, die auf dem *Host* beziehungsweise auf der CPU läuft, entweder ein 1D-, ein 2D- oder 3D-Gitter von

parallelen *Workgroups* und entweder ein 1D-, ein 2D- oder ein 3D-Gitter von parallelen *Workitems* innerhalb jeder Workgroup angeben (siehe Abbildung 3.4 für ein beispielhaftes 2D-Kernel) und dieses Kernel in eine der Task-Queues der GPU einreihen. Dadurch spannt das Kernel einen parallelen 1D-, 2D- oder 3D-Iterationsraum auf, weshalb wir im Folgenden ein Kernel, je nach der Dimensionalität seines Iterationsraums, als *1D-Kernel*, *2D-Kernel* oder *3D-Kernel* bezeichnen.

Auf der GPU arbeitet der Task-Scheduler seine Task-Queues asynchron zu der Host-Anwendung ab. Handelt es sich bei dem nächsten auszuführenden Task um ein Kernel, so beginnt der Task-Scheduler damit, die Workgroups des Kernels den Kernen der GPU zuzuweisen. Während eine Workgroup immer von nur einem Kern abgearbeitet wird, so kann ein Kern, je nach dem Ressourcenverbrauch einer Workgroup, mehrere Workgroups gleichzeitig abarbeiten. Jedes Workitem in einem Kernel entspricht einer Ausführung der Kernelfunktion, und kann zudem sowohl seinen *lokalen Index* innerhalb des Iterationsraums seiner Workgroup als auch seinen *globalen Index* innerhalb des Iterationsraums des Kernels abfragen. Dadurch kann es seinen eigenen Kontrollfluss innerhalb des Kernels beschreiten. Dabei besitzen sowohl der lokale Index als auch der globale Index für jede Dimension des Iterationsraums eine Komponente.

In einer Workgroup arbeitet ein jeder Thread per SIMD jeweils der Kontrollvektorbreite entsprechend viele aufeinanderfolgende Workitems ab, wobei es ein $1:1$ Mapping zwischen Workitems und den SIMD-Lanes dieses Threads gibt. Aus diesem Grund folgt die Vektorisierung bereits aus dem Quelltext des Kernels und der Kontrollvektorbreite der GPU, und jede Operation im Quelltext eines Kernels, die der Compiler nicht zu einer skalaren Operation optimieren kann, wird zu einer SIMD-Instruktion übersetzt (siehe Abbildung 3.5).

Zusätzlich arbeitet ein GPU-Kern via Multi-Threading sämtliche Threads einer Workgroup parallel zueinander ab. Dadurch können die Threads einer Workgroup über den Scratchpad-Speicher untereinander Daten austauschen und zusätzlich Daten aus dem Arbeitsspeicher über den geteilten L1-Cache untereinander wiederverwenden. Ebenso können sich die Threads einer Workgroup untereinander über eine spezielle Hardware-Barriere, die sämtliche Threads der Workgroup umspannt, synchronisieren. Deshalb ist es bei einem GPU-Kernel oft vorteilhaft, wenn die Threads einer Workgroup an derselben Aufgabe oder an benachbarten Aufgaben arbeiten, falls sie dadurch viele Daten untereinander wiederverwenden oder Zwischenergebnisse untereinander teilen können.

Die Preemption einer Workgroup, das heißt, dass ein GPU-Kern deren Abarbeitung vorzeitig beendet, findet auf GPUs nur in Sonderfällen, wie bei dem rekursiven Starten eines Kernels von einer Workgroup heraus, statt. Jedoch unterstützen GPUs dieses Preemption nicht auf eine Art und Weise, dass auf GPUs eine signalbasierte Synchronisation und Kommunikation zwischen zwei beliebigen Workgroups eines Kernels möglich wäre.

3.6 Kontrollfluss

In einem Kernel kann ein jedes Workitem seinen eigenen Kontrollfluss besitzen. Dadurch können aber auch die Workitems beziehungsweise die SIMD-Lanes eines Threads jeweils einen unterschiedlichen Kontrollfluss beschreiten. Dieses Phänomen wird als *Divergenz* der SIMD-Lanes eines Threads bezeichnet. SIMD-Architekturen wie GPUs verwenden zwei unterschiedliche grundlegende Konzepte für ihren Kontrollfluss:

Abbildung 3.4: Iterationsraum eines 2D-Kernels. In diesem Beispiel werden 2×2 Workgroups gestartet, wobei jede Workgroup wiederum 4×4 Workitems beinhaltet. Zusätzlich besitzt in diesem Beispiel ein GPU-Thread vier SIMD-Lanes, so dass ein jeder GPU-Thread vier aufeinanderfolgende Workitems gleichzeitig bearbeitet. Die Zahlen in der Abbildung geben die Indexe der Workgroups in ihrem Iterationsraum beziehungsweise die lokalen Indexe der Workitems innerhalb des Iterationsraums ihrer Workgroup an.

Kernelquelltext

```
code:
parfor each workgroup wg
   parfor each workitem wi in wg
      float a = // ...
      float b = // ...

      float c = a + b;
      float d = a - b;

      int i = get_global_index_x(wi);
      int j = // ...
      int k = i + j;

      // ...
```

Compiler-generierter GPU-Code

```
.vector registers:
   v_a, v_b, v_c, v_d,
   v_i, v_j, v_k;

.code:


   v_a = // ...
   v_b = // ...

   v_fadd v_c = v_a + v_b
   v_fsub v_d = v_a - v_b

   get_global_index_x v_i
   v_j = // ...
   v_iadd v_k = v_i + v_j

   // ...
```

Abbildung 3.5: Vektorisierung von arithmetischen Operationen in einem Kernel. Für die Vektorisierung übersetzt der GPU-Compiler jede arithmetische Operation im Quelltext in eine passende Vektorinstruktion im Maschinencode. Zudem muss der GPU-Compiler die beiden äußeren parallelen Iterationen über die Workgroups und Workitems des Kernels nicht von dem Quelltext in den Maschinencode übersetzen, da diese beiden parallelen Iterationen automatisch von dem Task-Scheduler auf der GPU beim Starten des Kernels beziehungsweise beim Verteilen der Workgroups auf die GPU-Kerne übernommen werden.

- **Sprungbefehle:** Sprungbefehle verändern den Befehlszähler eines Threads um oder auf einen gegebenen Wert, wodurch sich auch die nächste auszuführende Instruktion dementsprechend ändert.
- **Predication:** Beim *Predication* wird eine SIMD-Instruktion zusätzlich mit einem Predication-Register maskiert, weshalb diese Technik auch als *Masking* bezeichnet wird. Dieses Predication-Register enthält für jede SIMD-Lane eines Threads ein Bit. Ist das Bit gesetzt, so führt das Vektorrechenwerk die Operation der Instruktion für die entsprechende SIMD-Lane aus. Ist das Bit nicht gesetzt, dann bleibt das Vektorrechenwerk für die entsprechende SIMD-Lane untätig. Das Maskieren einer Instruktion via Predication kann sowohl explizit geschehen, das heißt in der Instruktion wird zusätzlich ein Predication-Register angegeben, als auch implizit, das heißt für die Maskierung wird ein Predication-Register verwendet, das in der Instruktion selbst nicht angegeben ist.

Die Abbildungen 3.6 und 3.7 zeigen, wie ein GPU-Compiler mit diesen beiden Konzepten einen if-else-Block sowie eine Schleife über mehrere Workitems hinweg vektorisiert.

Es gibt zwei unterschiedliche Ansätze, wie GPU-Architekturen diese beiden grundlegenden Konzepte umsetzen, um für jedes Workitem und damit für jede SIMD-Lane innerhalb eines Threads einen eigenen Kontrollfluss zu ermöglichen:

Kernelquelltext

```
code:
parfor each wg
   parfor each wi in wg
      float a = // ...
      float b = // ...

      if
         (a < b)


         a = a + b;

      else



         a = a - b;

      else_end:
      // ...
```

Compiler-generierter GPU-Code

```
.vector registers: v_a, v_b, v_c;
.predicate registers: p_0, p_1;
.code:


   v_a = // ...
   v_b = // ...

if_start:
   v_fcomp_less p_0 = v_a < v_b
   p_comp_eql_jump p_0 == 0 -> else_start

   v_fadd_masked v_a = v_a + v_b ? p_0

else_start:
   p_invert p1 = !p0
   p_comp_eql_jump p_1 == 0 -> else_end

   v_fsub_masked v_a = v_a - v_b ? p_1

else_end:
// ...
```

Abbildung 3.6: Vektorisierung eines if-else-Blocks in einem Kernel. Dabei gilt, dass die Instruktion `v_fcomp_less` beide Vektorargumentregister komponentenweise vergleicht und jeweils das entsprechende Bit im Predicate-Register setzt, falls der Vergleich wahr ist.

Kernelquelltext

```
code:
parfor each wg
   parfor each wi in wg
      float a = // ...
      float b = // ...
      float c = // ...

      while
         (c < b){


         c = c + a;

         }

      while_end:
      // ...
```

Compiler-generierter GPU-Code

```
.vector registers: v_a, v_b, v_c;
.predicate registers: p_0;
.code:


   v_a = // ...
   v_b = // ...
   v_c = // ...

while_start:
   v_fcomp_less p_0 = v_c < v_b
   p_comp_eql_jump p_0 == 0 -> while_end

   v_fadd_masked v_c = v_c + v_a ? p_0

   jump while_start

while_end:
   // ...
```

Abbildung 3.7: Vektorisierung einer while-Schleife in einem Kernel.

- **GPU-Architekturen mit einem Befehlszähler per Thread (AMD GPUs, NVIDIA GPUs vor Volta):** Tritt bei einer GPU mit einer solchen Architektur im Kontrollfluss eines Kernels eine Verzweigung auf, so überprüft die GPU für jeden Zweig, ob mindestens eine SIMD-Lane im Thread diesen Zweig nimmt. Wenn dies eintritt, dann verändert die GPU den Befehlszähler des Threads per Sprungbefehl so, dass er den Basic-Block des Zweigs ausführt, und sämtliche SIMD-Lanes des Threads, die diesen Zweig nicht ausführen, werden per Predication während der Ausführung dieses Zweiges deaktiviert. Wenn keine der SIMD-Lanes im Thread einen Zweig nimmt, dann verändert die GPU den Befehlszähler des Threads per Sprungbefehl so, dass er den Basic-Block des Zweigs überspringt. Während manche GPU-Architekturen dieses Vorgehen vollständig per Software implementieren, so implementieren andere GPU-Architekturen dieses Vorgehen per Hardware mit impliziter Predication.
- **GPU-Architekturen mit einem Befehlszähler per SIMD-Lane (NVIDIA GPUs seit Volta):** Die GPUs solcher Architekturen manipulieren bei einem Sprungbefehl die Befehlszähler jeder SIMD-Lane. Im Falle von Divergenz führt die GPU die Instruktionen der SIMD-Lanes beider Zweige Out-Of-Order aus, wobei sie die Instruktionen der SIMD-Lanes innerhalb eines Zweiges weiterhin In-Order ausführt. Zusätzlich deaktiviert die GPU jeweils diejenigen SIMD-Lanes, die in einem Zweig nicht aktiv sind, per impliziter Predication. Falls die Verzweigung im Kontrollfluss wieder konvergiert, so kann ein Kernel die Befehlszähler aller SIMD-Lanes eines Threads durch eine spezielle Kontrollinstruktion ebenfalls wieder synchronisieren.

Auf GPUs mit beiden Architekturen führt die Divergenz eines Threads dazu, dass sie die SIMD-Lanes dieses Threads via Predication deaktivieren müssen, und die Rechenwerke der deaktivierten SIMD-Lanes während der Ausführung des Zweiges untätig sind. Insbesondere gilt, dass in einem Kernel bei einem If-Block die vollen Kosten anfallen, sobald er nur von einer SIMD-Lane des Threads betreten wird, unabhängig davon, ob die anderen SIMD-Lanes den If-Block ebenfalls betreten. Ebenso führt ein Thread in einem Kernel eine Schleife so lange aus, bis sämtliche seiner SIMD-Lanes diese Schleife verlassen haben. Dabei fallen für jede Schleifeniteration die vollen Kosten an, unabhängig davon, wie viele SIMD-Lanes in einer Schleifeniteration noch aktiv sind.

3.7 Speicherräume

GPUs besitzen mehrere Speicherräume:

- **Heap-Speicher:** Der *Heap-Speicher* liegt im Arbeitsspeicher beziehungsweise im DRAM der GPU und wird für Daten verwendet, die zwischen zwei Kernel-Aufrufen persistent sein sollen. Zusätzlich beherrschen moderne GPU-Architekturen für ihren Heap-Speicher *Paging*, das heißt, sie können die Seiten ihres Heaps bei Speichermangel in den Arbeitsspeicher des Hosts auslagern, und bei einem Seitenfehler automatisch die Seite aus dem Speicher des Hosts anfordern. Ebenso kann eine GPU von einem Kernel heraus auf den Heap-Speicher einer anderen GPU zugreifen, wodurch diese GPUs dann eine Art RDMA-System bilden. Der Heap-Speicher kann sowohl von dem Host-Programm als auch von einem Kernel heraus alloziert werden.

- **Stack-Speicher:** Der *Stack-Speicher* liegt ebenfalls im Arbeitsspeicher beziehungsweise im DRAM der GPU und wird für die Stacks der Threads der GPU verwendet. Deshalb lebt der Stack-Speicher so lange wie der dazugehörige Thread. Analog wie bei CPUs wird der Stack-Speicher bei GPUs für Folgendes verwendet:
  - für *Register-Spilling*, das heißt das Auslagern von Registern auf den Stack, wenn zu wenig Register vorhanden sind, um alle automatischen Variablen eines Kernels abzuspeichern,
  - für automatische Arrays, die dynamisch indexiert werden, da der Registersatz nicht dynamisch indexiert werden kann, und
  - für das Sichern von Registern bei Funktionsaufrufen.

  Im Gegensatz zu CPUs ist der Stack-Speicher privat für die jeweilige SIMD-Lane, wodurch ihn GPU-Architekturen potentiell effizienter in den Caches ohne den Overhead der Cache-Kohärenz zwischenspeichern können. Auf Grund der kleinen Caches und vielen parallelen Threads sind Zugriffe auf den Stack sehr teuer. Deshalb sollten Kernels möglichst viele Daten über die Register wiederverwenden und Zugriffe auf den Stack vermeiden.

- **Scratchpad-Speicher:** Der Speicherraum des *Scratchpad-Speichers* repräsentiert denjenigen Scratchpad-Speicher, der für eine Workgroup auf einem GPU-Kern alloziert worden ist und worüber die Workitems einer Workgroup miteinander kommunizieren können. Diejenige Menge an Scratchpad-Speicher, die für jede Workgroup eines Kernels alloziert werden soll, kann sowohl zur Compile-Zeit des Kernels als auch zum Kernelstart festgelegt werden. Der Scratchpad-Speicher lebt so lange wie die dazugehörige Workgroup.

- **Konstantenspeicher:** Der *Konstantenspeicher*, sofern die GPU-Architektur ihn implementiert, liegt ebenfalls im Arbeitsspeicher beziehungsweise im DRAM der GPU. Er kann von einem Kernel aus nur gelesen werden, wird gesondert zwischengespeichert und für Skalar-Vektorinstruktionen verwendet.

Zusätzlich können GPUs potentiell von einem Kernel heraus auf zwei unterschiedliche Arten von Host-Speicher zugreifen:

- **Pinned Speicher:** Ein Zugriff von einem Kernel auf *pinned Speicher*, das heißt Speicher, dessen Seiten die CPU nicht auslagern darf, ist seit mittlerweile über 10 Jahren auf allen GPU-Architekturen ohne Einschränkungen möglich.

- **Pageable Speicher:** Ein Zugriff von einem Kernel auf *pageable Speicher*, das heißt Speicher, dessen Seiten die CPU auslagern darf, unterliegt auf den meisten aktuellen GPU-Architekturen noch diversen Einschränkungen. So muss unter anderem die Allozierung dieses pageable Speichers nicht per über die Standardfunktion `malloc`, sondern über die verwendete GPGPU-API, wie zum Beispiel in CUDA über die Funktion `cudaMallocManaged`, vorgenommen werden. Unseres Wissens ist ein allgemeiner Zugriff auf über die Standardfunktion `malloc` allozierten Host-Speicher gegenwärtig nur auf aktuellen NVIDIA-GPUs mit einer Power7-CPU als Host und einem NVLink-Bus als Verbindung zwischen GPU und Host möglich.

Diese einzelnen Speicherräume, mit Ausnahme des Konstantenspeichers, befinden sich zudem in einem vereinigten Speicherraum. Dadurch kann die GPU anhand einer Speicheradresse den spezifischen Speicherraum bestimmen, um dann von dem dazugehörigen Speicher die Daten anzufordern.

3.8 Speicherzugriffe

Es existieren zwei unterschiedliche Typen von SIMD-Speicherzugriffsinstruktionen:

- **SIMD-Block-Leseinstruktionen und SIMD-Block-Schreibinstruktionen (siehe Abbildung 3.8):** Während *SIMD-Block-Lese-Instruktionen* einen Block von Daten beginnend ab einer gegebenen Speicheradresse in ein gegebenes Vektorregister laden, schreiben *SIMD-Block-Schreibinstruktionen* den Inhalt eines gegebenen Vektorregisters beginnend ab der gegebenen Speicheradresse in den Speicher.
- **Gather-Instruktionen und Scatter-Instruktionen (siehe Abbildung 3.9):** Diese Instruktionen benötigen neben einem Vektorregister für die Daten ein Vektorregister mit Speicheradressen als zusätzliches Argument. Bei der Abarbeitung einer *Gather-Instruktion* werden nun die Daten an der $[i]$ten Adresse des Adressregisters in die $[i]$te Komponente des Datenregisters geladen, während bei einer *Scatter-Instruktion* die $[i]$te Komponente des Datenregisters an die $[i]$te Adresse des Adressregisters geschrieben wird.

In den meisten Kernels spielen Gather- und Scatter-Operationen auf Grund ihrer größeren Flexibilität die wichtigste Rolle. Deshalb werden Gather- und Scatter-Instruktionen von allen aktuellen GPU-Architekturen sehr effizient implementiert, während SIMD-Block-Leseinstruktionen und SIMD-Block-Schreibinstruktionen nicht einmal von allen aktuellen GPU-Architekturen unterstützt werden. Die Vektorisierung eines Speicherzugriffs mit einer Gather-Instruktion wird beispielhaft von Abbildung 3.10 gezeigt.

Die Abarbeitung einer Gather- oder Scatter-Instruktion auf einer GPU unterscheidet sich, je nachdem, auf welchen Speicher sie zugreift:

- **Auf den Scratchpad-Speicher:** Hier bestimmen die Load-Store-Einheiten, auf welche Scratchpad-Bank jede SIMD-Lane zugreift, und fordern die Daten dann von der entsprechenden Bank an. Wenn jedoch mehrere SIMD-Lanes auf unterschiedliche Adressen innerhalb derselben Bank zugreifen, verursacht dies *Bank-Konflikte*, die wiederum eine Sequentialisierung des Zugriffs verursachen.
- **Auf den Arbeitsspeicher:** Auf den ersten Blick scheinen Gather- und Scatter-Instruktionen eine effiziente Vektorisierung von feingranularen Zugriffen auf den Arbeitsspeicher zu sein. Jedoch sind feingranulare Zugriffe auf den Arbeitsspeicher sehr ineffizient, da sowohl die Caches und der DRAM jeweils nur komplette Cache-Lines übertragen können. Moderne GPU-Architekturen lindern dieses Problem mit einer Technik, die als *Coalescing* bezeichnet wird: Greifen mehrere SIMD-Lanes in einer Gather- oder Scatter-Instruktion auf dieselbe Cache-Line zu, so werden diese Zugriffe zu einem einzigen Zugriff auf diese Cache-Line verschmolzen (coalesced), wodurch wiederum

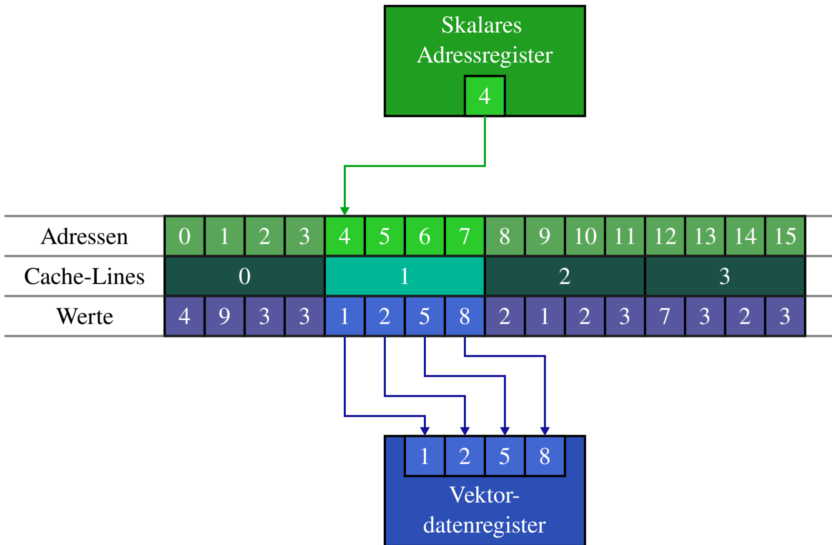


Abbildung 3.8: Zugriff auf den Arbeitsspeicher mit einer SIMD-Blockleseinstruktion.

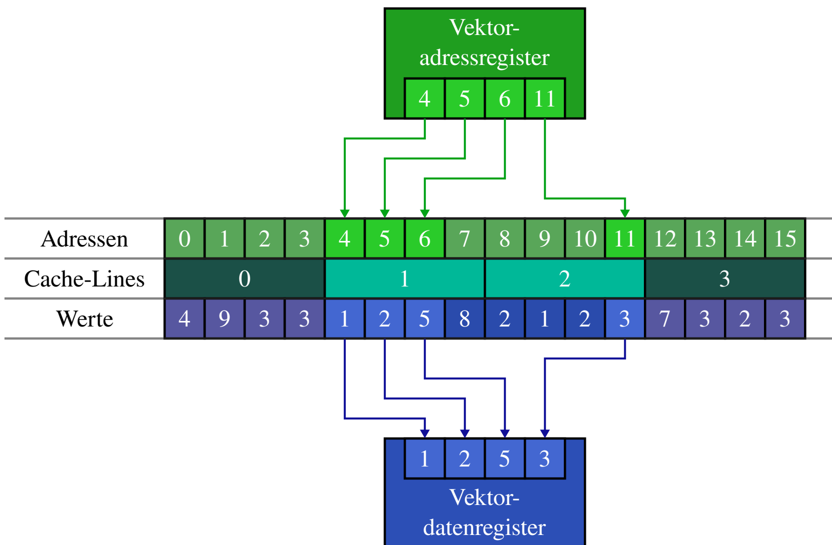


Abbildung 3.9: Zugriff auf den Arbeitsspeicher mit einer Gather-Instruktion. Auf Grund von Coalescing fordert ein GPU-Kern die Cache-Lines mit den Nummern 1 und 2 nur je einmal an, so dass diese auch nur einmal übertragen werden.

Kernelquelltext

```
in: float* A




.code:
parfor each wg
   parfor each wi in wg

      int i = get_global_id._x(wi);

      float* A_reg = A;

      float* a_i_ptr = A_reg  + i;


      float a_i_reg = *a_i_ptr;

      // ...
```

Compiler-generierter GPU-Code

```
.vector registers: v_i,
      v_a_i_ptr, v_a_i;
.scalar registers: s_A;
.data: float* A;

.code:



   get_global_index_x v_i

   s_load_ptr s_A = A

   v_imul v_i = 4 * v_i
   v_iadd v_a_i_ptr = s_A + v_i

   v_gather v_a_i = *v_a_i_ptr

   // ...
```

Abbildung 3.10: Vektorisierung eines Speicherzugriffs mit einer Gather-Instruktion in einem Kernel.

Cache-Bandbreite eingespart wird. Wenn eine Gather- oder Scatter-Instruktion sämtliche Daten in einer Cache-Line ausnutzt, so werden dank Coalescing keine übertragenen Daten verschwendet, wodurch Gather- und Scatter-Instruktionen in dieser Hinsicht genauso effizient sind, wie SIMD-Block-Lese- und SIMD-Block-Schreibinstruktionen.

GPUs implementieren Gather- und Scatter-Instruktionen unterschiedlicher Breite (meist 1, 2, 4, 8 und 16 Byte), welche pro SIMD-Lane einen Block der entsprechenden Größe aus dem Speicher lesen beziehungsweise in den Speicher schreiben. Da ein einzelnes Vektorregister für jede SIMD-Lane nur vier Byte bereithält, benötigen 1-Byte-, 2-Byte- und 4-Byte-Gather-Instruktionen nur ein Vektorregister, um die zu lesenden Daten abzuspeichern, während 8-Byte-Gather-Instruktionen zwei Vektorregister und 16-Byte-Gather-Instruktionen vier Vektorregister benötigen. Analoges gilt für die Anzahl an Vektorregister, die eine Scatter-Instruktion mit einer bestimmten Breite in den Speicher schreibt. Dies hat zur Folge, dass 8-Byte-Datentypen nach dem Laden über eine 8-Byte-Gather-Instruktion bereits über zwei Vektorregister verteilt vorliegen. So können sie auch ohne weiteres Shuffling direkt von den Vektorrechenwerken verwendet werden. Analoges gilt für das Zurückschreiben solcher Datentypen via 8-Byte-Scatter-Instruktionen.

3.9 Latenzüberbrückung

GPU-Architekturen implementieren zwei unterschiedliche Mechanismen, um die Latenzen bei Datenabhängigkeiten zwischen den Instruktionen eines Kernels zu überbrücken:

- **Ausnutzung der Parallelität auf Instruktionsebene via In-Order-Pipelines:** Auf GPUs arbeitet eine solche In-Order-Pipeline die Instruktionen eines Threads gemäß ihrer Reihenfolge im Maschinencode des Kernels ab. Allerdings kann die Abarbeitung der nächsten Instruktion schon beginnen, bevor die Ergebnisse der vorherigen Instruktionen verfügbar sind, sofern keine Datenabhängigkeiten zwischen diesen Instruktionen bestehen. Falls ein Operand für die nächste auszuführende Instruktion jedoch noch nicht verfügbar ist, entsteht ein Datenkonflikt. Dieser Datenkonflikt blockiert den auszuführenden Thread so lange, bis der Operand verfügbar wird. Dieses parallele Ausführen von Instruktionen, sofern dies die Datenabhängigkeiten erlauben, wird als *Parallelität auf Instruktionsebene* (auch *Instruction Level Parallelism* oder *ILP* genannt) bezeichnet.
- **Ausnutzung der Parallelität auf Thread-Ebene via Fine-Grained Interleaved Multithreading:** Eine GPU kann die *Parallelität auf Thread-Ebene* (auch *Thread Level Parallelism* oder *TLP* genannt) via *Fine-Grained Interleaved Multithreading* ausnutzen. Dieses Multi-Threading ermöglicht es dem Thread-Scheduler, wenn einer seiner Threads wegen eines Datenkonflikts in seiner Pipeline blockiert ist, zwischenzeitlich die Instruktionen von anderen Threads, welche nicht blockiert sind, ausführen zu lassen. Dadurch können die Rechenwerke vollständig ausgelastet werden, wenn in jedem Takt nur ein einziger Thread bereit ist, seine nächste Instruktion auszuführen. Zudem können sich dadurch die Instruktionen von unterschiedlichen Threads gleichzeitig in den In-Order-Pipelines eines GPU-Kerns befinden.

3.10 Multithreading mit einer variablen Anzahl von parallelen Threads

Auf modernen GPU-Architekturen ist das Register-Spilling wegen deren kleinen Caches sehr teuer. GPU-Architekturen von NVIDIA und AMD lindern die Kosten des Register-Spillings, indem auf ihnen die Anzahl an Registern pro Thread beziehungsweise die Anzahl an Threads pro Subkern kein fester Wert ist, sondern der GPU-Compiler kann für ein Kernel eine bestimmte Anzahl an Registern pro Thread allozieren. So kann ein GPU-Compiler das Register-Spilling für ein Kernel minimieren, indem er für dieses Kernel eine große Anzahl an Registern pro Thread alloziert. Allerdings verringert sich für ein Kernel mit einer großen Anzahl an Registern pro Thread auch die maximale Anzahl an parallel ausgeführten Threads pro Subkern. Dadurch kann ein Subkern bei einer großen Anzahl an Registern pro Thread nur wenige Threads gleichzeitig ausführen, wodurch er wiederum durch über das Fine-Grained Interleaved Multithreading die Latenzen schlechter überbrücken kann. Der Zusammenhang zwischen der Anzahl an Registern pro Thread und der Anzahl an nebenläufig ausgeführten Threads pro Subkern wird beispielhaft für die NVIDIA-Volta-Architektur von der Tabelle 3.2 gezeigt.

Standardmäßig alloziert ein GPU-Compiler für einen Thread eines Kernel immer ausreichend viele Register, so dass kein Register-Spilling auftritt. Erst wenn die maximal mögliche, durch die Hardware limitierte Anzahl an Registern für einen Thread erreicht ist, fügt der Compiler gezwungenermaßen Register-Splling in das Kernel ein. Ein Programmierer besitzt aber zusätzlich die Möglichkeit, die Anzahl an Registern pro Thread mit der Hilfe von Compiler-Optionen auf einen benutzerdefinierten maximalen Wert zu limitieren. Hierdurch erhöht der Programmierer die Parallelität auf Thread-Ebene auf Kosten von mehr Register-Spilling.

Wegen dieser variablen Anzahl an parallelen Threads kann ein Programmierer bei vielen Kernels auch eine Feinabstimmung zwischen der Parallelität auf Instruktionsebene und der Parallelität auf Thread-Ebene vornehmen (siehe zum Beispiel [T20] für eine experimentelle Un-

# Vregs. pro Thread	32	40	48	56	64	72	80	96	128	168	255
# Thrds. pro Subkern	16	12	10	9	8	7	6	5	4	3	2

Tabelle 3.2: Anzahl an simultanen ausgeführten Threads pro Subkern in Abhängigkeit von der Anzahl an Registern pro Thread auf einer Volta-GPU. Die Werte in dieser Tabelle ergeben sich dadurch, dass ein Subkern 512 Vektorregister besitzt, maximal 16 Threads gleichzeitig verwalten kann, er zudem für einen Thread nur ein Vielfaches von 8 Registern und maximal 255 Register allozieren kann.

tersuchung hiervon). Für diese Feinabstimmung muss der Programmierer einem Workitem des Kernels mehr Aufgaben zuweisen, und das Kernel zusätzlich so gestalten, dass keine unnötigen Datenabhängigkeiten zwischen den Aufgaben vorliegen. Dadurch kann der Compiler einen Maschinencode mit einer höheren Parallelität auf Instruktionsebene erzeugen. Für diese höhere Parallelität auf Instruktionsebene muss der Compiler jedoch für jeden Thread mehr Register allozieren, wodurch sich auch die Anzahl an nebenläufigen Threads pro Subkern und die Parallelität auf Thread-Ebene verringert.

3.11 Occupancy

GPU-Terminologien bezeichnen diejenige Anzahl an Threads, die ein Kern einer GPU bei einem bestimmten Kernel gleichzeitig ausführen kann, als *Occupancy*. Die Occupancy ist insgesamt von vielen Hardware-Beschränkungen der gegebenen GPU-Architektur abhängig. Im Wesentlichen gelten folgende Regeln:

- Jeder Kern kann maximal nur eine bestimmte Anzahl an Workgroups verwalten.
- Es können jeweils nur vollständige Workgroups in einem GPU-Kern Platz finden.
- Ein Kernel kann für seine Workgroups den Scratchpad-Speicher nur in einer bestimmten Granularität allozieren.
- Ein Kernel kann maximal nur eine bestimmte Menge an Scratchpad-Speicher für eine seiner Workgroup allozieren; weniger Scratchpad-Speicher als in einem GPU-Kern vorhanden ist.
- Jeder GPU-Kern besitzt nur eine beschränkte Menge von Scratchpad-Speicher. Dadurch kann er effektiv nur so viele Workgroups gleichzeitig ausführen, wie in seinem Scratchpad-Speicher Platz finden.
- Jede Workgroup darf nur eine bestimmte maximale Anzahl an Workitems enthalten.
- Jeder Subkern kann nur eine bestimmte maximale Anzahl an Threads gleichzeitig ausführen.
- Der GPU-Compiler kann für ein Kernel nur wenige bestimmte Werte für die Anzahl an Registern pro Thread festlegen.
- Jeder Subkern besitzt nur eine beschränkte Anzahl von Registern, und kann effektiv nur so viele Threads gleichzeitig ausführen, wie in seinem Registersatz Platz finden.

Da diese Regeln sehr komplex und spezifisch für die jeweilige GPU-Architektur sind, stellen GPU-Hersteller den Entwicklern oft Werkzeuge oder APIs für die Berechnung der Occupancy zur Verfügung.

3.12 Task-Queues und Events

Der *Task-Scheduler* auf der GPU besitzt mehrere *Task-Queues*, welche auszuführende Tasks beinhalten. Bei einem solchen Task kann es sich um ein auszuführendes Kernel oder um eine Kopieroperation handeln. Dabei arbeitet der Task-Scheduler die Task-Queues parallel zueinander ab, so dass sich die Berechnungen von mehreren Kernels sowie eine Kopieroperation mit den Berechnungen eines Kernels überlappen können. Ebenso wird der CPU-Thread der Host-Anwendung bei dem Einreihen eines Kernels in eine Task-Queue nicht blockiert, sondern er kann mit seinen Berechnungen fortfahren und weitere Kernel in die Task-Queues der GPU einreihen. Des Weiteren können die Task-Queues sowohl *Events* auslösen als auch auf Events warten. Mit Hilfe dieser Events ist es möglich sowohl zwei Task-Queues untereinander als auch eine Task-Queue mit einem CPU-Thread zu synchronisieren.

3.13 Programmierung von GPUs über GPU-APIs

Es gibt eine Vielzahl von GPU-APIs für die Programmierung von GPUs. Bekannte Vertreter hierfür sind CUDA [W9], OpenCL [W10], C++ AMP [T21], DirectX [W11], Vulkan [W12] und OpenGL [W13] oder hip [W14]. In all diesen modernen GPU-APIs lassen sich die meisten in diesem Kapitel erläuterten Grundkonzepte wiederfinden. Auf Grund der unterschiedlichen Terminologien wird ein Grundkonzept von einer GPU-API allerdings meist anders bezeichnet als in dieser Arbeit. Es gibt zudem Implementierungen für GPU-APIs, welche es ermöglichen, ein Kernel nicht nur parallel auf einer GPU, sondern auch auf einer CPU (siehe Abschnitt 13.1 wie zum Beispiel OpenCL für CPUs implementiert wird) oder einem anderen Beschleuniger, wie einem FPGA, auszuführen. GPU-APIs ermöglichen die Programmierung einer GPU-beschleunigten Anwendung über zwei unterschiedliche Ansätze:

- **Getrennter Quelltext:** Der klassische Ansatz für die Programmierung einer GPU-beschleunigten Anwendungen basiert auf einer Trennung des Quelltextes. Durch diese Trennung wird der CPU-Code und GPU-Code jeweils in separaten Quelltextdateien abgespeichert und separat kompiliert. Dabei findet die Compilierung des GPU-Codes in der Regel zur Laufzeit der GPU-beschleunigten Anwendung statt. Bei diesem Ansatz ist es auch leicht möglich, dass die GPU-beschleunigte Anwendung den GPU-Code nicht aus einer Datei lädt, sondern ihn selbst dynamisch erzeugt, wodurch sie diesen GPU-Code auch leicht zu ihrer Laufzeit auf eine spezifische GPU-Architektur optimieren kann.
- **Geteilter Quelltext:** Bei diesem moderneren Ansatz für die Programmierung von GPU-beschleunigten Anwendungen wird eine Hochsprache, wie Fortran, C oder C++, mit GPU-spezifischen Erweiterungen versehen. Zusätzlich darf eine Quelltextdatei sowohl GPU-Code als auch CPU-Code beinhalten. Dadurch kann ein Programmierer sowohl im CPU-Code als auch im GPU-Code auf die gleichen Strukturen, Klassen und Funktionen zurückgreifen, wodurch sich die Programmierung der GPU-beschleunigten Anwendung stark vereinfacht. Allerdings wird dann bei der Kompilierung der GPU-beschleunigten Anwendung nicht nur ein regulärer Compiler der Hochsprache benötigt, sondern eine spezielle Toolchain. Bei dieser Toolchain entfernt zunächst eine Art von Präprozessor sämtlichen GPU-Code aus den Quelltextdateien, und leitet diesen entfernten GPU-Code an einen GPU-Compiler weiter. Nach der Anwendung des Präprozessors befindet sich in diesen Quelltextdateien ein Programm in der reinen Hochsprache, welches dann von ei-

nem regulären Compiler der Hochsprache übersetzt werden kann. Dadurch ist es bei diesem Ansatz auch nicht möglich, dass eine GPU-beschleunigte Anwendung ihren GPU-Code während ihrer Ausführung selbst erzeugt oder verändert.

Während OpenCL nur den Ansatz des getrennten Quelltexts verfolgt, so ermöglicht CUDA sowohl den Ansatz mit dem getrennten Quelltext über die *CUDA-Driver-API* als auch den Ansatz mit dem geteilten Quelltext über die *CUDA-Runtime-API*.

3.14 Vergleich zwischen modernen GPU-Architekturen und modernen x64-CPU-Architekturen

Da eine Vielzahl an modernen CPU-Architekturen mit einem weit gefächerten Anwendungsbereich existiert, fällt ein allgemeiner Vergleich zwischen modernen GPU-Architekturen und CPU-Architekturen schwer. Deshalb werden wir uns im Folgenden für diesen Vergleich auf moderne x64-CPU-Architekturen aus dem Server-Bereich, wie die Skylake-Architektur von Intel oder die Ryzen-Architektur von AMD, beschränken, da wir diese Architekturen auch als Ziel für unsere Optimierungen und für unsere Benchmarks verwenden werden. Generell gilt jedoch, dass viele der Aussagen in dem folgenden Vergleich auch für andere CPU-Architekturen aus dem Server- oder HPC-Bereich ihre Gültigkeit bewahren. Dabei stützt sich dieser folgende Vergleich auf die CPU-Spezifikationen in den Quellen [T22], [T23] und [W15]. Die Daten von konkreten CPUs mit solchen modernen x64-Server-Architekturen sind in Abschnitt 13.6 gelistet. Für diese Auflistung dieser CPU-Daten wählten wir eine ähnliche Struktur wie für die Auflistung der GPU-Daten in Abschnitt 13.5, so dass ein Leser die Daten von beiden Architekturen leicht miteinander vergleichen kann. So besitzen moderne GPU-Architekturen und moderne x64-Server-CPU-Architekturen viele grundlegende Gemeinsamkeiten:

- **Ähnlicher Grundaufbau:** Moderne x64-Server-CPUs und GPUs lassen sich beide im Groben in Speicherinterface, geteilte Caches und Rechenkerne gliedern. Auch ist der Aufbau von einem GPU-Kern einer modernen GPU und einem CPU-Kern einer modernen x64-Server-CPU im Groben ähnlich. Beide besitzen private Caches, einen Thread-Scheduler, einen skalaren Registersatz sowie einen Vektorregistersatz und mehrere unterschiedliche skalare Rechenwerke sowie Vektorrechenwerke.
- **Predication sowie Gather- und Scatter-Instruktionen:** Moderne x64-Server-CPU-Architekturen mit der AVX-512-Befehlssatzerweiterung, wie zum Beispiel die Skylake-Architektur von Intel, beherrschen genauso wie moderne GPU-Architekturen, Predication sowie Gather- und Scatter-Instruktionen. Dadurch können Programme auf solchen CPUs ein analoges Vorgehen wie auf GPUs nutzen, um per Software beziehungsweise per Predication mehrere Kontrollflüsse innerhalb eines Hardware-Threads mit SIMD-Instruktionen abzuarbeiten. Während ein Coalescing bei Gather- und Scatter-Instruktionen auch auf CPU-Architekturen prinzipiell möglich ist, so fanden wir leider keine Dokumentation darüber, inwieweit moderne CPU-Architekturen tatsächlich ein solches Coalescing implementieren. Zumindest gemäß dem Buch [B6] wird dieses Coalescing für Gather-Instruktionen auf der Knights-Corner-Architektur für Xeon-PHI-Beschleunigerkarten von Intel implementiert, während es allerdings für die nachfolgende Knights-Landing-Architektur wieder verworfen wurde. Moderne x64-Server-CPUs, die nicht die AVX-512-, sondern nur die AVX2-Befehlssatzerweiterung implementieren, wie

zum Beispiel die Ryzen-Architektur von AMD, beherrschen zwar Gather-Instruktionen, jedoch weder Predication noch Scatter-Instruktionen.

So bestehen zwischen modernen GPU-Architekturen und modernen x64-Server-CPU-Architekturen auch viele Unterschiede:

- **Unterschiede im Aufbau:** Bei modernen GPU-Architekturen sind das Speicherinterface und die Vektorinstruktionen deutlich breiter, der Vektorregistersatz deutlich größer, die Cache-Hierarchie weniger komplex und die Caches deutlich kleiner als auf modernen x64-Server-CPU-Architekturen. Zudem besitzen die Caches auf modernen GPUs im Gegensatz zu den Caches auf modernen x64-Server-CPUs keinen Hardware-Prefetcher. Während die feinere Gliederung eines GPU-Kerns in Subkerne bei allen modernen GPU-Architekturen auftritt, ist sie zwar auch bei modernen x64-Server-CPU-Architekturen prinzipiell möglich, wie zum Beispiel bei der Bulldozer-Architektur von AMD, aber sehr selten.
- **Unterschiede beim Scheduling von Tasks:** Auf modernen GPU-Architekturen weist ein spezieller Hardware-Task-Scheduler die Aufgaben beziehungsweise die Workgroups den Kernen zu, während auf CPUs das Betriebssystem dafür verantwortlich ist, die Kernelthreads beziehungsweise die Prozesse den Hardwarethreads der CPU-Kerne zuzuweisen. Ebenso arbeitet eine GPU ihre Workgroups weitestgehend ohne Preemption ab, wodurch keine signalbasierte Synchronisation zwischen zwei beliebigen Workgroups eines Kernels möglich ist. Dahingegen kann das Betriebssystem die Threads auf der CPU jederzeit per Preemption beziehungsweise per Interrupts unterbrechen, und darüber signalbasierte Synchronisation zwischen zwei beliebigen CPU-Threads ermöglichen.
- **Unterschiede beim Multithreading und beim Pipelining:** Während ein GPU-Kern via fine-grained-interleaved Multithreading viele Threads gleichzeitig ausführen kann (bis zu 64 auf modernen NVIDIA GPUs), kann ein CPU-Kern einer modernen x64-Server-CPU via simultaneous Multithreading nur zwei Threads gleichzeitig ausführen. Dafür arbeitet ein CPU-Kern einer modernen x64-Server-CPU seine Threads in einer komplexen Out-Of-Order Pipeline ab, welche die Latenzen besser überbrücken kann, als die einfache In-Order Pipeline eines GPU-Kerns. Des Weiteren kooperieren auf GPUs bei vielen GPU-Kernels die Threads einer Workgroup an einer Aufgabe, können sich leicht über eine Hardware-Barriere synchronisieren und Daten über den Scratchpad-Speicher teilen. Dahingegen arbeiten auf x64-Server-CPUs bei den meisten Programmen die parallelen Hardware-Threads eines Kerns an unterschiedlichen Aufgaben. Dies beruht darauf, dass auf einer modernen x64-Server-CPU eine Synchronisation über eine Hardware-Barriere zwischen den Threads eines Kerns und ein Teilen von Daten zwischen den Threads eines Kerns über einen Scratchpad-Speicher nicht möglich ist, und zusätzlich das Betriebssystem jeden Hardware-Thread als einen eigenständigen logischen Kern behandelt. Moderne x64-Server-CPUs implementieren zudem Register-Renaming, welches die physikalischen Register eines CPU-Kerns kompetitiv zur Laufzeit eines Programms unter den nebenläufigen Threads eines Kerns aufteilt. Dahingegen alloziert der GPU-Compiler für ein GPU-Kernel eine feste Anzahl von Registern pro Thread, welche sich während der Laufzeit des GPU-Kernels nicht ändert.
- **Unterschiede bei der logischen Vektorbreite und bei dem Predication:** Auf modernen x64-Server-CPUs verändert sich die logische Vektorbreite, also wie viele Vektorkompo-

nenten mit einer SIMD-Instruktion verarbeitet werden können, je nach Größe des Datentyps, so dass die Anzahl an Bits in einem Vektor konstant ist. Dadurch ist auf solchen x64-Server-CPUs die logische Vektorbreite für einen 4-Byte-Datentypen doppelt so groß wie für einen 8-Byte-Datentypen. Zusätzlich erwarten die Vektorrechenwerke auf modernen x64-Server-CPUs im Gegensatz zu den Vektorrechenwerken auf einer GPU jedoch, dass ein Vektor eines 8-Byte-Datentyps sich nur in einem Vektorregister befindet, und nicht wie auf GPUs über zwei Vektorregister verteilt vorliegt.

- **Unterschiede bei den Gather- und Scatter-Instruktionen:** Moderne x64-Server-CPUs beherrschen ebenfalls Gather- und Scatter-Instruktionen mit einer Breite von 4 Byte und 8 Byte. Da die Vektorrechenwerke auf einer modernen x64-Server-CPU erwarten, dass ein Vektor eines 8-Byte-Datentyps sich nur in einem Vektorregister befindet, befüllt eine 8-Byte-Gather-Instruktion auch nur ein einziges Vektorregister mit Daten und eine 8-Byte-Scatter-Instruktion schreibt nur ein einziges Vektorregister mit Daten zurück. Dabei transferiert auf einer modernen x64-Server-CPU eine 8-Byte-Gather- oder 8-Byte-Scatter-Instruktion jedoch nur halb so viele Blöcke wie eine 4-Byte-Gather- oder 4-Byte-Scatter-Instruktion. Im Gegensatz dazu befüllt auf einer modernen GPU eine 8-Byte-Gather-Instruktion zwei Vektorregister mit Daten und eine 8-Byte-Scatter-Instruktion schreibt zwei Vektorregister mit Daten zurück.
- **Unterschiedliches Verhalten bei einem Write-Miss im Last-Level-Cache:** Auf modernen GPUs implementiert der L2-Cache als Last-Level-Cache eine Write-Allocate-Policy mit No-Fetch-On-Write, wodurch die GPU bei einem L2-Write-Miss die Cache-Line zwar im L2-Cache alloziert, jedoch keine Daten aus dem DRAM lädt. Auf modernen x64-Server-CPUs implementiert dahingegen der L3-Cache als Last-Level-Cache eine Write-Allocate-Policy mit Fetch-On-Write, wodurch die CPU bei einem L3-Write-Miss die Cache-Line im L3-Cache alloziert und dabei sofort die Daten der Cache-Line aus dem DRAM anfordert. Somit verursacht auf x64-Server-CPUs ein Write-Miss im Last-Level-Cache einen höheren Overhead als auf GPUs. Es ist allerdings auf modernen x64-Server-CPUs über sogenannte *Non-Temporal-Store-Instruktionen*, welche oft auch *Streaming-Store-Instruktionen* bezeichnet werden, möglich, Daten direkt in den DRAM zu schreiben, wodurch dieser Overhead entfällt.

3.15 Fazit

In diesem Kapitel beschrieben wir den Aufbau und die Funktionsweise von modernen GPU-Architekturen. Dabei zeigten wir, dass GPUs sehr viele Konzepte implementieren, die auch von anderen Prozessoren wie CPUs bekannt sind. Ebenso zeigten wir, dass sich moderne GPU-Architekturen hinreichend gut mit der gebräuchlichen CPU-Terminologie beschreiben lassen und dass keine spezielle GPU-Terminologie nötig ist.

4 Grundlagen zu ODE-Verfahren

4.1 Allgemeines

Wie bereits in der Einleitung erwähnt, so betrachten wir in dieser Arbeit *Anfangswertprobleme* (englisch: *initial value problem*, abgekürzt *IVP*) von Systemen von *gewöhnlichen Differentialgleichungen* (englisch: *ordinary differential equation*, abgekürzt: *ODE*):

$$\textbf{Gegeben:} \quad \mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)), \quad t_0, \quad \mathbf{y}(t_0) = \mathbf{y}_0, \quad t_e \tag{4.1}$$

$$\textbf{Gesucht:} \quad \mathbf{y}_e = \mathbf{y}(t_e) \tag{4.2}$$

Bei einem ODE-System bestehend aus n Gleichungen gilt, dass $\mathbf{y} \in \mathbb{R}^n$, $t \in \mathbb{R}$ sowie $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n$ ist. Der klassische numerische Lösungsansatz, welcher auch von den *ODE-Verfahren* in dieser Arbeit angewendet wird, basiert auf einer Zeitschrittprozedur. Diese Prozedur beginnt zur Simulationszeit t_0 mit einem Anfangswert $\mathbf{y}_0$ und führt eine Folge von Zeitschritten $t_\kappa \rightarrow t_{\kappa+1}$ mit $\kappa = 0, 1, 2, \ldots$ aus, bis sie die finale Zeit t_e der Simulation erreicht hat. Bei jedem Zeitschritt κ wendet die Zeitschrittprozedur die *Funktion der rechten Seite* (englisch: *right-hand-side*, abgekürzt: *RHS*) an, um einen neuen Zustand der Simulation $\mathbf{y}_{\kappa+1}$ zu berechnen, der die exakte Lösung der Funktion $\mathbf{y}(t)$ zum Zeitpunkt $t_{\kappa+1}$ approximiert. Eine ausgiebige Erläuterung der in diesem Kapitel vorgestellten Grundlagen von ODE-Verfahren kann zum Beispiel in den Büchern [B7–B9] gefunden werden.

4.2 Klassifikation von ODE-Verfahren

Im Folgenden werden wir kurz auf die Unterteilung von ODE-Verfahren in e*xplizite ODE-Verfahren* und in *implizite ODE-Verfahren* sowie auf deren Unterteilung in *Einschrittverfahren* und *Mehrschrittverfahren* eingehen, da beides für unsere Datenflussgraphräpresentation relevant ist. Die Einteilung von ODE-Verfahren in Einschritt- und Mehrschrittverfahren erfolgt anhand folgender Kriterien:

- **Einschrittverfahren:** Ein Zeitschritt κ in einem *Einschrittverfahren* verwendet nur den aktuellen Systemzustand $\mathbf{y}_\kappa$ und potentiell dessen zeitliche Ableitung $\mathbf{y}'_\kappa$, welches beides vom vorherigen Zeitschritt $\kappa - 1$ berechnet wurde, um den nächsten Systemzustand $\mathbf{y}_{\kappa+1}$ zu berechnen. Innerhalb eines Zeitschritts kann ein Einschrittverfahren über mehrere Stufen Zwischenzustände des Systems und deren zeitliche Ableitungen berechnen, wodurch wiederum die Genauigkeit des Einschrittverfahrens erhöht wird.
- **Mehrschrittverfahren:** Ein Zeitschritt κ in einem *Mehrschrittverfahren* verwendet nicht nur den aktuellen Systemzustand $\mathbf{y}_\kappa$ aus dem letzten Zeitschritt $\kappa - 1$, sondern auch weitere Systemzustände und zeitliche Ableitungen (unter anderem $\mathbf{y}_\kappa, \ \ldots \ , \mathbf{y}_{\kappa-k-1}$ sowie $\mathbf{y}'_\kappa, \ \ldots \ , \mathbf{y}'_{\kappa-k-1}$) aus den k letzten Zeitschritten $\kappa - 1, \ \ldots \ , \kappa - k$, um den nächsten Systemzustand $\mathbf{y}_{\kappa+1}$ zu berechnen. Somit benötigt ein Mehrschrittverfahren ein Einschrittverfahren als Startprozedur, welche aus dem Anfangswert $\mathbf{y}_0$ alle benötigten Systemzustände und benötigten zeitlichen Ableitungen berechnet. Je nachdem auf wie viele k-vergangenen Zeitschritte ein Mehrschrittverfahren für die Berechnung des aktuellen Zeitschritts zugreift, wird es auch als ein k-Schrittverfahren bezeichnet.

Sowohl Einschritt- als auch Mehrschrittverfahren lassen sich danach klassifizieren, ob sie explizit oder implizit sind:

- **Explizite ODE-Verfahren:** Die mathematische Definition von einem *expliziten ODE-Verfahren* definiert einen neuen Systemzustand nur über vergangene beziehungsweise bereits bekannte Systemzustände und deren zeitliche Ableitungen. Aus diesem Grund kann bei expliziten ODE-Verfahren deren mathematische Definition bereits als Berechnungsvorschrift des Verfahrens dienen.
- **Implizite ODE-Verfahren:** Die mathematische Definition von einem *impliziten ODE-Verfahren* definiert mindestens einen neuen Systemzustand nicht nur über vergangene und damit bereits bekannte Systemzustände und deren zeitliche Ableitungen, sondern auch über zukünftige und damit noch unbekannte Systemzustände und deren zeitliche Ableitungen. Somit gibt es in der mathematischen Definition eines impliziten ODE-Verfahrens eine zyklische Abhängigkeit zwischen den Systemzuständen und deren zeitlichen Ableitungen. Dadurch kann bei einem impliziten Verfahren die mathematische Definition nicht mehr als Berechnungsvorschrift dienen, sondern der Zeitschritt eines impliziten ODE-Verfahrens erfordert das Lösen eines nichtlinearen Gleichungssystems.

Dabei gilt, dass explizite ODE-Verfahren für nicht-steife Anfangswertprobleme besser geeignet sind, während implizite ODE-Verfahren für steife Anfangswertprobleme besser geeignet sind (siehe Abschnitt 5.1).

4.3 Parallelität in ODE-Verfahren

Als Nächstes werden wir kurz auf die Arten von Parallelität in einem ODE-Verfahren eingehen, welche die Implementierung eines Lösers für Anfangswertprobleme ausnutzen kann:

- **Systemparallelität:** Bei den in dieser Arbeit betrachteten ODE-Verfahren setzt sich ein Zeitschritt aus Auswertungen der RHS-Funktion, Linearkombinationen, Map-Funktionen und Reduktionen zusammen. Offensichtlich kann ein Prozessor bei einer Linearkombination, einer Map-Funktion oder einer Reduktion die einzelnen Systemkomponenten parallel zueinander berechnen. Diese feingranulare Art der Parallelität von ODE-Verfahren wird allgemein von der Literatur als *Systemparallelität* bezeichnet. Zusätzlich besitzt bei den allermeisten relevanten Anfangswertproblemen die RHS-Funktion ebenfalls einen hohen Grad an Systemparallelität. Ebenso tritt diese Systemparallelität bei Lösern für nichtlineare Gleichungssysteme, die für implizite ODE-Verfahren benötigt werden, wie zum Beispiel dem Newton-Verfahren, auf. Des Weiteren bietet die Systemparallelität bei Linearkombinationen, Map-Funktionen, Reduktionen sowie bei vielen RHS-Funktionen auch einen sehr hohen oder sogar einen perfekten Grad an Datenparallelität. Aus diesen Gründen lässt sich die Systemparallelität nicht nur dafür ausnutzen, um die Berechnung der Komponenten des ODE-Systems auf mehrere Rechenkerne oder Prozessoren aufzuteilen, sondern auch dafür, um über die Komponenten des ODE-Systems zu vektorisieren.
- **Methodenparallelität:** Bei einigen ODE-Verfahren, wie zum Beispiel bei den PIRK-Verfahren, den PEER-Verfahren oder auch bei diversen RK-Verfahren, berechnet ein

Zeitschritt mehrere Auswertungen der RHS-Funktion, Linearkombinationen, MAP-Funktionen oder Reduktionen, welche voneinander unabhängig sind. Auf Grund ihrer Unabhängigkeit voneinander, können diese Berechnungen auch parallel zueinander durchgeführt werden. Diese Art der Parallelität von ODE-Verfahren wird von der Literatur als *Methodenparallelität* bezeichnet. Auf Grund ihrer groben Granularität wurde die Methodenparallelität in der Literatur (siehe zum Beispiel [B8]) ursprünglich nur als eine Art von Task-Parallelität aufgefasst. Für das Ausnutzen dieser Task-Parallelität wurden die voneinander unabhängigen Auswertungen der RHS-Funktion, Linearkombinationen, Map-Funktionen oder Reduktionen jeweils als ein eigenständiger Task einem oder mehreren Prozessoren zur Berechnung zugewiesen. Oft beinhaltet die Methodenparallelität aber auch einen gewissen Grad an Datenparallelität, wodurch man sie auch zur Vektorisierung der Berechnungen ausnutzen kann. Selbst wenn die Methodenparallelität ursprünglich nicht dafür entworfen wurde, so zeigten neuere Untersuchungen, wie wir sie zum Beispiel in der Arbeit [E1] durchführten, dass sich die Methodenparallelität auch hervorragend dafür eignet, um die Speicherzugriffslokalität und darüber die Performance einer Implementierung eines ODE-Verfahrens zu erhöhen. Dabei zeigen unsere Messungen in der Arbeit [E1] sogar, dass das Ausnutzen der Methodenparallelität zur Erhöhung der Speicherzugriffslokalität weit performanter sein kann, als die Methodenparallelität, für ihren ursprünglichen Zweck, also als eine Form von Task-Parallelität, zu verwenden.

- **Zeitparallelität:** Bei manchen ODE-Verfahren, wie zum Beispiel bei den Parareal-Verfahren, führt die Zeitschrittprozedur nacheinander mehrere Folgen von Zeitschritten durch, wobei diese Folgen zum Teil voneinander unabhängig sind. Dadurch können diese Folgen parallel zueinander berechnet werden. Diese Art der Parallelität wird als *Zeitparallelität* bezeichnet. Analog zur Methodenparallelität lässt sich die Zeitparallelität als Taskparallelität auffassen. Über diese Abstraktion der Taskparallelität werden die voneinander unabhängigen Zeitschrittfolgen jeweils als ein eigenständiger Task interpretiert, und jeweils einem oder mehreren Prozessoren zur Berechnung zugewiesen.

4.4 Lokaler Diskretisierungsfehler und Ordnung

Ein ODE-Verfahren kann für jeden Zeitschritt nur eine Approximation des nächsten Systemzustands berechnen. Dadurch verursacht jeder Zeitschritt also, verglichen mit der exakten Lösung des Zeitschritts ausgehend von dessen aktuellen Systemzustand, einen Fehler, welcher als *lokaler Diskretisierungsfehler* bezeichnet wird. Angenommen, ein ODE-Verfahren berechnet bei einem Zeitschritt κ mit der Schrittweite h aus der aktuellen Approximation des Systemzustands $\mathbf{y}_\kappa$ bei t_κ die nächste Approximation des Systemzustands $\mathbf{y}_{\kappa+1}$ bei $t_\kappa + h$. Zudem sei $\mathbf{y}_{\kappa+1,\text{exakt}}$ die exakte Lösung des Zeitschritts ausgehend von $\mathbf{y}_\kappa$, das heißt, die exakte Lösung des Anfangswertproblems mit $\mathbf{y}_\kappa$ als Startwert, t_κ als Startzeit und $t_\kappa + h$ als Endzeit. Dann ist der lokale Diskretisierungsfehler $\epsilon_{\kappa+1}$ des Zeitschritts κ definiert als:

$$\epsilon_{\kappa+1} = ||\mathbf{y}_{\kappa+1} - \mathbf{y}_{\kappa+1,\text{exakt}}||. \tag{4.3}$$

Die Ordnung eines ODE-Verfahrens ist nun definiert, als derjenige größtmögliche Wert von p, der mit einer beliebig kleinen Konstante $C > 0$ folgende Ungleichung erfüllt:

$$\epsilon_{\kappa+1} \leq Ch^p. \tag{4.4}$$

ODE-Verfahren können diesen lokalen Diskretisierungsfehler für einen Zeitschritt abschätzen, indem sie zusätzlich zur Lösung $\mathbf{y}_{\kappa+1}$ eines Zeitschritts κ eine weitere Lösung $\hat{\mathbf{y}}_{\kappa+1}$ mit einer geringeren Ordnung berechnen. Über die Differenz zwischen der eigentlichen Lösung und der Lösung mit der geringeren Ordnung berechnen sie dann einen Fehlervektor $\mathbf{E}$:

$$\mathbf{E} = (\mathbf{y}_{\kappa+1} - \hat{\mathbf{y}}_{\kappa+1}) \oslash \mathbf{S}, \tag{4.5}$$

mit dem verfahrensspezifischen Skalierungsfaktoren $\mathbf{S}$, wobei der $\oslash$-Operator die Hadamard-Division, das heißt die komponentenweise Division von dem Vektor auf der linken Seite durch den Vektor auf der rechten Seite, ist. Diesen Fehlervektor reduzieren ODE-Verfahren dann mit einer Vektornorm zu dem geschätzten lokalen Diskretisierungsfehler:

$$\epsilon_{\kappa+1} \approx \|\mathbf{E}\|, \tag{4.6}$$

wobei ODE-Verfahren als Vektornorm meist die euklidische Norm oder die Maximumsnorm verwenden.

4.5 Schrittweitensteuerung

Sowohl Einschritt- als auch Mehrschrittverfahren können mit einer *Schrittweitensteuerung* kombiniert werden, die die Schrittweite anpasst und Zeitschritte verwirft, so dass für jeden Zeitschritt ein benutzerdefinierter lokaler Diskretisierungsfehler nicht überschritten wird.

Für eine Schrittweitensteuerung wird am Ende eines jeden Zeitschritts κ zunächst ein lokaler Fehler $\epsilon_{\kappa+1}$ wie im vorherigen Abschnitt 4.4 beschrieben abgeschätzt. Als Nächstes überprüft die Schrittweitensteuerung am Ende des Zeitschritts, ob sich diese Fehlerabschätzung innerhalb einer benutzerdefinierten Toleranz ϵ_{max} befindet. Wenn ja, also falls $\epsilon_{\kappa+1} \leq \epsilon_{\text{max}}$, wird der nächste Zeitschritt berechnet, wenn nein, also falls $\epsilon_{\kappa+1} > \epsilon_{\text{max}}$, wird der aktuelle Zeitschritt verworfen und erneut berechnet. Zusätzlich bestimmt die Schrittweitensteuerung eine neue Schrittweite h_{neu} für die Berechnung des nächsten Zeitschritts $\kappa + 1$ beziehungsweise für die Wiederholung des aktuellen Zeitschritts κ. Auf Grund ihrer Funktionsweise lässt sich die Schrittweitensteuerung als ein Regler in einem Regelkreis interpretieren, weshalb man die Schrittweitensteuerung eines ODE-Verfahrens, wie es beispielsweise in der Arbeit [Z29] geschehen ist, auch mit Hilfe der Regelungstheorie untersuchen und erklären kann. Deshalb kann eine Schrittweitensteuerung für die Bestimmung der neuen Schrittweite prinzipiell einen beliebigen Regler aus der Regelungstechnik verwenden. Dieser Regler sollte die Schrittweite so groß wie möglich wählen, aber zudem sicherstellen, dass möglichst wenige Zeitschritte verworfen werden müssen. Hierfür eignet sich ein *PI-Regler*, wie er in der Arbeit [Z29] vorgeschlagen wurde, besonders gut, weshalb ein solcher auch meist für ODE-Verfahren verwendet wird:

$$h_{\text{neu}} = \left(\frac{1}{\epsilon_{\text{neu}}}\right)^{\alpha} \left(\frac{1}{\epsilon_{\text{alt}}}\right)^{\beta} h_{\text{alt}}, \tag{4.7}$$

wobei h_{neu} die als Nächstes zu verwendende Schrittweite, h_{alt} die zuletzt verwendete Schrittweite, ϵ_{neu} die aktuelle Approximation des lokalen Diskretisierungsfehlers, sowie ϵ_{alt} die zuvor ermittelte Approximation des lokalen Diskretisierungsfehlers ist. Zusätzlich sind $\alpha \in \mathbb{R}^+$ und $\beta \in \mathbb{R}^-$ zwei Konstanten, die passend zum Anfangswertproblem, dem ODE-Verfahren sowie dessen Ordnung zu wählen sind.

Soll ein Mehrschrittverfahren mit einer Schrittweitensteuerung kombiniert werden, so befinden sich die vergangenen Systemzustände und zeitlichen Ableitungen, die das Mehrschrittverfahren als zusätzliche Stützstellen verwendet, nicht mehr an denjenigen Zeitpunkten, zu welchen das Mehrschrittverfahren diese erwartet. Somit müssen diese vergangenen Systemzustände und zeitlichen Ableitungen nach einer Änderung der Schrittweite an den benötigten Zeitpunkten interpoliert beziehungsweise extrapoliert werden, so dass das Mehrschrittverfahren das Ergebnis der Interpolation beziehungsweise Extrapolation als Stützstellen verwenden kann.

4.6 Explizite RK-Verfahren

Die allgemeinste Klasse expliziter Einschrittverfahren ist die Klasse der *expliziten RK-Verfahren*, deren allgemeine Beschreibung in der Arbeit [Z30] formuliert wurde. Diese umfassen auch die Extrapolationsverfahren und die expliziten iterierten RK-Verfahren.

Zu jedem Zeitschritt κ berechnet ein explizites RK-Verfahren aus einer *Eingabeapproximation* des Systemzustands $\mathbf{y}_\kappa$ über eine Folge von dazwischenliegenden *s (internen) Stufen* eine *Ausgabeapproximation* des Systemzustands $\mathbf{y}_{\kappa+1}$.

Jede Stufe $i = 1, \dots, s$ berechnet dabei zunächst einen Zwischenzustand des Systems $\mathbf{Y}_i$ am Zeitpunkt $t_\kappa + h_\kappa c_i$ mit einer Linearkombination, um dann aus diesem Zwischenzustand eine zeitliche Ableitung $\mathbf{F}_i$ via der Auswertung der RHS-Funktion zu ermitteln. Die Linearkombination des Zwischenzustands $\mathbf{Y}_i$ einer Stufe i setzt sich dabei aus dem Vektor der Eingabeapproximation $\mathbf{y}_\kappa$ und den zeitlichen Ableitungen $\mathbf{F}_1, \dots, \mathbf{F}_{i-1}$ aller vorherigen Stufen $1, \dots, i-1$ mit den verfahrensspezifischen Gewichten $a_{i1}, \dots, a_{i(i-1)}$ zusammen. In mathematische Formeln gefasst gilt:

$$\mathbf{Y}_i = \mathbf{y}_\kappa + h_\kappa \sum_{j=1}^{i-1} a_{ij}\mathbf{F}_j, \tag{4.8}$$

$$\mathbf{F}_i = \mathbf{f}(t_\kappa + h_\kappa c_i, \mathbf{Y}_i). \tag{4.9}$$

Dabei ist $A = (a_{ij}) \in \mathbb{R}^{s\times s}$ eine verfahrensspezifische strikte untere Dreiecksmatrix und $\mathbf{c} \in \mathbb{R}^s$ ein verfahrensspezifischer Koeffizientenvektor. Zudem entartet die Linearkombination für die erste Stufe zu $\mathbf{Y}_1 = \mathbf{y}_\kappa$, wodurch sie für diese erste Stufe entfällt.

Als Letztes berechnet ein RK-Verfahren die Ausgabeapproximation des Systemzustands $\mathbf{y}_{\kappa+1}$ per Linearkombination aus der Eingabeapproximation $\mathbf{y}_\kappa$ und den zeitlichen Ableitungen $\mathbf{F}_1, \dots, \mathbf{F}_s$ der Stufen mit den verfahrensspezifischen Gewichten $\mathbf{b} \in \mathbb{R}^s$:

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h_\kappa \sum_{j=1}^{s} b_j\mathbf{F}_j. \tag{4.10}$$

Sogenannte eingebettete RK-Verfahren berechnen, um effizient einen lokalen Diskretisierungsfehler für die Schrittweitensteuerung zu ermitteln, zunächst eine eingebettete Lösung $\hat{\mathbf{y}}_{\kappa+1}$ wiederum per Linearkombination aus der Eingabeapproximation $\mathbf{y}_\kappa$ und den zeitlichen Ableitungen $\mathbf{F}_1, \dots, \mathbf{F}_s$ der Stufen mit den zusätzlichen verfahrensspezifischen Gewichten $\hat{\mathbf{b}} \in \mathbb{R}^s$:

$$\hat{\mathbf{y}}_{\kappa+\mathbf{1}} = \mathbf{y}_\kappa + h_\kappa \sum_{j=1}^{s} \hat{b}_j\mathbf{F}_j. \tag{4.11}$$

Aus der Differenz zwischen der eingebetteten Lösung $\hat{\mathbf{y}}_{\kappa+1}$ und der Ausgabeapproximation $\mathbf{y}_{\kappa+1}$ lässt sich ein Fehlervektor $\mathbf{E}$ berechnen:

$$\mathbf{E} = \hat{\mathbf{y}}_{\kappa+1} - \mathbf{y}_{\kappa+1}. \tag{4.12}$$

Die Berechnung des Fehlervektors $\mathbf{E}$ lässt sich umformen zu:

$$\mathbf{E} = h_\kappa \sum_{j=1}^{s} (\hat{b}_j - b_j)\mathbf{F}_j. \tag{4.13}$$

Über den Fehlervektor kann ein eingebettetes RK-Verfahren leicht den lokalen Diskretisierungsfehler $\epsilon_{\kappa+1}$ abschätzen:

$$\epsilon_{\kappa+1} = \|\mathbf{E}\|. \tag{4.14}$$

Diesen lokalen Diskretisierungsfehler kann das RK-Verfahren anschließend für eine Schrittweitensteuerung verwenden.

Die Koeffizienten eines expliziten RK-Verfahrens lassen sich elegant und übersichtlich in einem sogenannten *Butcher-Tableau* darstellen:

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b}^T \\ & \hat{\mathbf{b}}^T \end{array} \quad . \tag{4.15}$$

Letztlich gilt anzumerken, dass für einige RK-Verfahren A, $\mathbf{b}$ und $\hat{\mathbf{b}}$ nicht vollbesetzt sind, wodurch sich die Abhängigkeiten zwischen den Berechnungen des Verfahrens reduzieren. Insbesondere kann es bei einem expliziten RK-Verfahren vorkommen, dass dieses Verfahren wegen zusätzlicher Nulleinträge in der Matrix A voneinander unabhängige Stufen besitzt. Bei solchen voneinander unabhängigen Stufen fließt das Ergebnis der einen Stufe nicht in die Berechnung der anderen Stufe ein und umgekehrt. Durch diese Unabhängigkeit können die Stufen parallel zueinander berechnet werden, wodurch in diesem RK-Verfahren Methodenparallelität auftritt.

Zusätzlich gilt anzumerken, dass bei vielen expliziten RK-Verfahren für alle Stufen $i = 1, \ldots, s$ die sogenannte *Knotenbedingung* erfüllt ist:

$$c_i = \sum_{j=1}^{i-1} a_{ij}. \tag{4.16}$$

Eine besondere Klasse von eingebetteten expliziten RK-Verfahren sind die sogenannten *FSAL-Verfahren* (*First-Same-As-Last-Verfahren*), bei welchem die Ausgangsapproximation $\mathbf{y}_{\kappa+1}$ eines Zeitschritts κ identisch mit dem Systemzustand $\mathbf{Y}_{\kappa,s}$ der letzten Stufe s dieses Zeitschritts ist, wodurch auch die zeitliche Ableitung $\mathbf{F}_{\kappa,s}$ der letzten Stufe eines Zeitschritts κ und die zeitliche Ableitung der ersten Stufe $\mathbf{F}_{\kappa+1,1}$ des nächsten Zeitschritts $\kappa + 1$ jeweils auf demselben Systemzustand berechnet werden. Hierfür muss bei FSAL-Verfahren zusätzlich gelten:

$$a_{sj} = \hat{b}_j, \quad j = 1, \ldots, s, \tag{4.17}$$

$$c_1 = 0, \tag{4.18}$$

$$c_s = 1. \tag{4.19}$$

Dies bewirkt wiederum, dass

$$\mathbf{Y}_{\kappa,s} = \mathbf{y}_{\kappa+1}, \tag{4.20}$$

$$\mathbf{F}_{\kappa,s} = \mathbf{f}t_\kappa + h_\kappa c_s, \mathbf{Y}_{\kappa,s}) = \mathbf{f}(t_\kappa + h_\kappa, \mathbf{y}_{\kappa+1}) = \mathbf{f}(t_{\kappa+1}, \mathbf{y}_{\kappa+1}) = \mathbf{F}_{\kappa+1,1} \tag{4.21}$$

ist. Dadurch lässt sich bei FSAL-Verfahren pro Zeitschritt eine Auswertung der RHS-Funktion einsparen.

Zusätzlich gilt, dass RK-Verfahren mit mehr Stufen auch potentiell eine höhere Ordnung besitzen als RK-Verfahren mit weniger Stufen. Dabei kann ein explizites RK-Verfahren mit s Stufen maximal eine Ordnung von $p = s$ erreichen.

4.7 Implizite RK-Verfahren

Implizite RK-Verfahren sind implizite Einschrittverfahren, deren allgemeine Formulierung von den Arbeiten [Z30] und [Z31] entwickelt wurde. Sie besitzen eine ähnliche mathematische Definition wie explizite RK-Verfahren, mit dem Unterschied, dass mindestens eine Stufe für ihre Linearkombination die zeitliche Ableitung einer noch unbekannten Stufe benötigt. Somit gibt es bei einem impliziten RK-Verfahren mindestens eine zyklische Abhängigkeit zwischen den Stufen, welche die Implizitheit des Verfahrens verursacht.

Zu jedem Zeitschritt κ berechnet ein implizites RK-Verfahren analog zu einem expliziten RK-Verfahren aus einer Eingabeapproximation des Systemzustands $\mathbf{y}_\kappa$ über mehrere dazwischenliegenden s Stufen eine Ausgabeapproximation des Systemzustands $\mathbf{y}_{\kappa+1}$. Dafür verwendet es die verfahrensspezifischen Gewichte A, $\mathbf{b}$, $\hat{\mathbf{b}}$, $\mathbf{c}$, die sich analog wie bei einem expliziten RK-Verfahren elegant in einem Butcher-Tableau darstellen lassen. Im Gegensatz zu einem expliziten RK-Verfahren darf in einem impliziten RK-Verfahren eine Stufe für die Berechnung ihres Zwischenzustands des Systems potentiell die zeitlichen Ableitungen aller Stufen in ihrer Linearkombination verwenden:

$$\mathbf{Y}_i = \mathbf{y}_\kappa + h_\kappa \sum_{j=1}^{s} a_{ij} \mathbf{F}_j, \tag{4.22}$$

$$\mathbf{F}_i = \mathbf{f}(t_\kappa + h_\kappa c_i, \mathbf{Y}_i), \tag{4.23}$$

wobei bei impliziten RK-Verfahren die Koeffizientenmatrix A mindestens einen Eintrag auf oder über der Diagonalen enthält, und auch vollbesetzt sein kann. Bei einem impliziten RK-Verfahren ist es zudem nicht möglich, sämtliche Einträge auf oder über der Diagonalen durch eine Permutation der Stufen zu eliminieren. Dadurch existiert bei einem impliziten RK-Verfahren mindestens eine zyklische Abhängigkeit zwischen zwei verschiedenen Stufen oder innerhalb einer Stufe. Durch diese zyklischen Abhängigkeiten kann die mathematische Definition der Stufen nicht mehr als Berechnungsvorschrift dienen. Auch kann die Implementierung eines impliziten RK-Verfahrens nicht mehr die Stufen ihrer Reihe nach berechnen, sondern sie muss für die Berechnung der Stufen ein nichtlineares Gleichungssystem zum Beispiel mit dem *Newton-Verfahren* lösen.

Abschließend berechnet ein implizites RK-Verfahren analog zu einem expliziten RK-Verfahren den nächsten Systemzustand $\mathbf{y}_{\kappa+1}$ und optional eine eingebettete Lösung $\hat{\mathbf{y}}_{\kappa+1}$ für die Abschätzung des lokalen Diskretisierungsfehlers.

Ein implizites RK-Verfahren mit s Stufen kann maximal eine Ordnung von $p \leq 2s$ erreichen. Somit kann ein implizites RK-Verfahren eine doppelt so hohe Ordnung erzielen wie ein explizites RK-Verfahren mit der gleichen Stufenzahl. Implizite RK-Verfahren, die für ihre Stufenzahl die maximale Ordnung von $p = 2s$ erreichen, sind zum Beispiel die Gauß-Verfahren.

4.8 PIRK-Verfahren

Parallele iterierte RK-Verfahren (*PIRK-Verfahren*) sind explizite Einschrittverfahren, die von den Arbeiten [Z1] und [Z2] mit dem Ziel einer hohen Methodenparallelität entwickelt wurden. Zudem lassen sie sich als eine Unterklasse der eingebetteten RK-Verfahren betrachten.

Ein PIRK-Verfahren wird aus einem impliziten RK-Verfahren, das als *Basisverfahren* bezeichnet wird, konstruiert. Dabei löst das PIRK-Verfahren die implizite Formulierung des Basisverfahrens beziehungsweise dessen zyklische Abhängigkeiten mit einem *Prädiktor-Korrektor-Schema* auf, so dass das PIRK-Verfahren selbst explizit wird. Im folgenden Abschnitt referenzieren wir die Stufenzahl des Basisverfahrens mit s, die Ordnung des Basisverfahrens mit p und dessen Koeffizienten mit A, $\mathbf{b}$ und $\mathbf{c}$.

Zuerst schätzt ein PIRK-Verfahren die zeitlichen Ableitungen der Stufen des Basisverfahrens $\mathbf{F}_1, \ldots, \mathbf{F}_s$ in einem *Prädiktorschritt* ab. Für diesen Prädiktorschritt gibt es viele mögliche Prädiktorschemata. Das einfachste Prädiktorschema, das wir auch in dieser Arbeit verwenden, setzt die zeitlichen Ableitungen aller Stufen mit der zeitlichen Ableitung der Eingabeapproximation gleich:

$$\mathbf{F}_i^{(0)} = \mathbf{f}(t_\kappa, \mathbf{y}_\kappa) = \mathbf{F}^{(0)}, \quad i = 1, \ldots, s. \tag{4.24}$$

Als Nächstes führt ein PIRK-Verfahren abhängig von der Ordnung p des Basisverfahrens eine feste Anzahl von $m = p - 1$ *Korrektorschritten* aus. Für jeden Korrektorschritt $k = 1, \ldots, p-1$ verfeinert das PIRK-Verfahren die zeitlichen Ableitungen der Stufen des Basisverfahrens $\mathbf{F}_1, \ldots, \mathbf{F}_s$ weiter. Dafür wandelt das PIRK-Verfahren die Stufenberechnung des Basisverfahrens leicht ab. Dieses leicht abgewandelte Stufenberechnung berechnet für jede Stufe i des Basisverfahrens per Linearkombination aus $\mathbf{y}_\kappa$ und den zeitlichen Ableitungen $\mathbf{F}_1^{(k-1)}, \ldots, \mathbf{F}_s^{(k-1)}$ des vorherigen Korrektorschritts $k-1$ beziehungsweise des Prädiktorschritts und den Gewichten $a_{i1}, \ldots, a_{is}$ einen neuen Systemzustand $\mathbf{Y}_i^{(k)}$. Auf diesen wird dann die RHS-Funktion angewendet, um für diese $i$te Stufe des Basisverfahrens eine neue verfeinerte zeitliche Ableitung des Systemzustands $\mathbf{F}_i^{(k)}$ zu erhalten. In mathematische Formeln gefasst gilt:

$$\mathbf{Y}_i^{(k)} = \mathbf{y}_\kappa + h_\kappa \sum_{j=1}^{s} a_{ij} \mathbf{F}_j^{(k-1)}, \quad i = 1, \ldots, s, \tag{4.25a}$$

$$\mathbf{F}_i^{(k)} = \mathbf{f}\left(t_\kappa + h_\kappa c_i, \mathbf{Y}_i^{(k)}\right), \qquad i = 1, \ldots, s. \tag{4.25b}$$

Dabei lässt sich Gleichung (4.25a) für den ersten Korrektorschritt vereinfachen zu:

$$\mathbf{Y}_i^{(1)} = \mathbf{y}_\kappa + h_\kappa (a_{i1} + \ldots + a_{is}) \mathbf{F}^{(0)}, \quad i = 1, \ldots, s. \tag{4.26}$$

Als Letztes berechnet ein PIRK-Verfahren die Ausgangsapproximation $\mathbf{y}_{\kappa+1}$ des Zeitschritts per Linearkombination von der Eingabeapproximation $\mathbf{y}_\kappa$ und den Funktionsauswertungen $\mathbf{F}_1^{(m)},\ \dots\ ,\mathbf{F}_s^{(m)}$ des finalen Korrektorschritts m mit den Gewichten $\mathbf{b}$ des Basisverfahrens:

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h_\kappa \sum_{j=1}^{s} b_j \mathbf{F}_j^{(m)}. \tag{4.27}$$

PIRK-Verfahren können zudem eine eingebettete Lösung über die zeitlichen Ableitungen des vorletzten Korrektorschritts $m-1$ ermitteln:

$$\hat{\mathbf{y}}_{\kappa+1} = \mathbf{y}_\kappa + h_\kappa \sum_{j=1}^{s} b_j \mathbf{F}_j^{(m-1)}. \tag{4.28}$$

Wie wir am Anfang dieses Abschnitts bereits erwähnten, so können wir PIRK-Verfahren als eine besondere Art der eingebetteten RK-Verfahren betrachten. Mit Hilfe der Koeffizienten A, $\mathbf{b}$, $\hat{\mathbf{b}}$ und $\mathbf{c}$ des Basisverfahrens können wir für ein PIRK-Verfahren folgendes Butcher-Tableau konstruieren:

$$\begin{array}{c|ccccc} 0 & & & & & \\ \mathbf{c} & A' & & & & \\ \mathbf{c} & 0 & A & & & \\ \vdots & \vdots & \ddots & \ddots & & \\ \mathbf{c} & 0 & \cdots & 0 & A & \\ \hline & 0^T & \cdots & 0^T & 0^T & \mathbf{b}^T \\ & 0^T & \cdots & 0^T & \mathbf{b}^T & 0^T \end{array}\ , \tag{4.29}$$

wobei gilt $A' = (\mathbf{a}', 0,\ \dots\ , 0) \in \mathbb{R}^{s\times s}$ mit $\mathbf{a}' = (\sum_{i=1}^{s} a_{1i}\ ,\ \dots\ ,\ \sum_{i=1}^{s} a_{si})^T \in \mathbb{R}^s$.

4.9 Peer-Verfahren

Im Gegensatz zu den eingebetteten RK-Verfahren verwenden die *Peer-Verfahren* aus der Arbeit [Z3] nicht nur mehrere *interne Stufen*, das heißt Stufen, welche der aktuelle Zeitschritt als Zwischenapproximationen berechnet, sondern auch mehrere *externe Stufen*, das heißt Zwischenapproximationen aus dem vorherigen Zeitschritt. Deshalb sind sie weder reine Einschrittverfahren noch reine Mehrschrittverfahren, aber gehören zur Oberklasse der allgemeinen linearen Verfahren und können als Zweischritt-RK-Verfahren betrachtet werden.

Zu jedem Zeitschritt κ berechnet ein Peer-Verfahren s Stufen, wobei es typischerweise zwischen zwei und acht Stufen in einem Peer-Verfahren gibt. Jede Stufe $i = 1, \dots, s$ approximiert einen Systemzustand $\mathbf{y}_{\kappa+1,i}$ bei $t_{\kappa+1,i} = t_{\kappa,i} + h_\kappa$. Alle diese Systemzustände besitzen dieselben Stabilitätseigenschaften und Genauigkeit. Um ihren neuen Systemzustand zu approximieren, benötigt eine Stufe die Systemzustände aller Stufen $\mathbf{y}_{\kappa,1},\ \dots\ ,\mathbf{y}_{\kappa,s}$ aus dem vorherigen Zeitschritt, die zeitlichen Ableitungen $\mathbf{F}_{\kappa,1},\ \dots\ ,\mathbf{F}_{\kappa,s}$ von diesen Zuständen, und die variable Schrittweite h_κ.

Zu Beginn des Zeitschritts κ berechnet ein Peer-Verfahren für jede Stufe $i = 1, \dots, s$ den neuen Systemzustand $\mathbf{y}_{\kappa+1,i}$ der Stufe über eine Linearkombination von den Systemzuständen

$\mathbf{y}_{\kappa,1}, \dots, \mathbf{y}_{\kappa,s}$ aller Stufen und den zeitlichen Ableitungen $\mathbf{F}_{\kappa,1}, \dots, \mathbf{F}_{\kappa,s}$ von diesen Zuständen:

$$\mathbf{y}_{\kappa+1,i} = \sum_{j=1}^{s} \left(b_{ij}\mathbf{y}_{\kappa,j} + h_\kappa a_{ij}\mathbf{F}_{\kappa,j} \right), \quad i = 1, \dots, s, \tag{4.30}$$

mit den verfahrensspezifischen Koeffizienten $A = (a_{ij}) \in \mathbb{R}^{s\times s}$ und $B = (b_{ij}) \in \mathbb{R}^{s\times s}$.

Als Zweites berechnet das Peer-Verfahren für jede Stufe $i = 1, \dots, s$ die neue zeitliche Ableitung $\mathbf{F}_{\kappa+1,i}$ über eine Auswertung der RHS-Funktion auf $\mathbf{y}_{\kappa+1,i}$ zu dem Zeitpunkt $t_{\kappa+1,i}$:

$$\mathbf{F}_{\kappa+1,i} = \mathbf{f}\left(t_{\kappa+1,i}, \mathbf{y}_{\kappa+1,i}\right). \tag{4.31}$$

Als Drittes berechnet das Peer-Verfahren einen Fehlervektor $\mathbf{E}$:

$$\mathbf{E} = \left| \sum_{l=1}^{s} e_l \mathbf{F}_{\kappa+1,l} \right|_{\text{komp}} \oslash \mathbf{S}, \tag{4.32}$$

mit den verfahrensspezifischen Koeffizienten $\mathbf{e} \in \mathbb{R}^s$, den Skalierungsfaktoren

$$\mathbf{S} = (\text{atoli}, \dots, \text{atoli})^T + \text{rtoli} \left|\mathbf{y}_{\kappa,\mathbf{1}}\right|_{\text{komp}}, \tag{4.33}$$

und benutzerdefinierten Toleranzen atoli und rtoli. Hierbei beschreibt $|\mathbf{x}|_{\text{komp}}$ den komponentenweisen Betrag eines Vektors, und der $\oslash$-Operator wieder die Hadamard-Division, das heißt die komponentenweise Division von dem Vektor auf der linken Seite durch den Vektor auf der rechten Seite.

Zuletzt schätzt das Peer-Verfahren den lokalen Diskretisierungsfehler ϵ über die Maximumsnorm des Fehlervektors ab:

$$\epsilon = \|\mathbf{E}\|_\infty. \tag{4.34}$$

Diesen lokalen Diskretisierungsfehler kann das Peer-Verfahren anschließend für eine Schrittweitensteuerung verwenden.

4.10 Adams-Bashforth-Verfahren

Adams-Bashforth-Verfahren sind explizite Mehrschrittverfahren, welche von der Arbeit [B10] entwickelt wurden. Für jeden Zeitschritt κ berechnet ein k-Schritt Adams-Bashforth-Verfahren den nächsten Systemzustand $\mathbf{y}_{\kappa+1}$ über eine Linearkombination des aktuellen Systemzustands $\mathbf{y}_\kappa$, der zeitlichen Ableitung $\mathbf{F}_\kappa$ dieses aktuellen Systemzustands und der zeitlichen Ableitungen $\mathbf{F}_{\kappa-1}, \dots, \mathbf{F}_{\kappa-k+1}$ der Systemzustände der letzten $k-1$ Zeitschritte. Aus dem Grund besteht ein Zeitschritt eines k-Schritt-Adams-Bashforth-Verfahrens aus einer Auswertung der RHS-Funktion gefolgt von einer Linearkombination:

$$\mathbf{F}_\kappa = \mathbf{f}(t_\kappa, \mathbf{y}_\kappa), \tag{4.35}$$

$$\mathbf{y}_{\kappa+1} = \mathbf{y}_\kappa + h \sum_{i=1}^{k} b_i \mathbf{F}_{\kappa-i+1}. \tag{4.36}$$

Dabei sind $\mathbf{b} = (b_i) \in R^k$ verfahrensspezifische Koeffizienten, die sich aus k als Gewichte für eine Polynominterpolation berechnen lassen.

4.11 Richardson-Extrapolationsverfahren

Bei dem *Richardson-Extrapolationsverfahren*, welches in den Arbeiten [Z32] und [Z33] entwickelt wurde, handelt es sich um einen Vertreter der Klasse der *Extrapolationsverfahren* (siehe [B9]). Diese Extrapolationsverfahren führen in jedem Zeitschritt (*Makroschritt*) mit einem gegebenen Einschrittverfahren (*Basisverfahren*) ausgehend vom aktuellen Zustand des Systems mehrere Folgen von Zeitschritten (*Mikroschritte*) durch, wobei sie für jede dieser Folgen von Mikroschritten eine unterschiedliche Schrittweite verwenden und dadurch auch wiederum für jede dieser Folgen unterschiedlich viele Mikroschritte durchführen müssen. Jede dieser Folgen von Mikroschritten liefert zudem eine eigene Lösung für den Makroschritt, woraus das Extrapolationsverfahren dann eine noch genauere Lösung für den Makroschritt extrapolieren kann.

Das Richardson-Extrapolationsverfahren berechnet zwei Folgen von Mikroschritten: Eine Folge mit der Schrittweite h, bei welcher es für den Makroschritt nur einen Mikroschritt benötigt, und eine weitere Folge mit der Schrittweite von $h/2$, bei welcher es für den Makroschritt zwei Mikroschritte benötigt. So führt es zunächst für die erste Folge ausgehend von dem aktuellen Systemzustand $\mathbf{y}_\kappa$ einen Mikroschritt mit einer Schrittweite von h durch, um den nächsten Systemzustand $\mathbf{y}^h_{\kappa+1}$ bei $t_\kappa + h$ zu ermitteln. Als Nächstes führt es für die zweite Folge ebenfalls ausgehend von $\mathbf{y}_\kappa$ zwei Mikroschritte mit der halbierten Schrittweite $h/2$ durch, um zunächst aus $\mathbf{y}_\kappa$ den Systemzustand $\mathbf{y}^{h/2}_{\kappa+1/2}$ bei $t_\kappa + h/2$ und anschließend daraus den Systemzustand $\mathbf{y}^{h/2}_{\kappa+1}$ bei $t_\kappa + h$ zu berechnen. Aus den beiden Systemzuständen $\mathbf{y}^{h/2}_{\kappa+1}$ und $\mathbf{y}^h_{\kappa+1}$ sowie aus der Ordnung p des Basisverfahrens kann das Richardson-Extrapolationsverfahren nun eine genauere Lösung $\mathbf{y}_{\kappa+1}$ für den Zeitschritt κ extrapolieren:

$$\mathbf{y}_{\kappa+1} = \mathbf{y}^{h/2}_{\kappa+1} + \frac{\mathbf{y}^{h/2}_{\kappa+1} - \mathbf{y}^h_{\kappa+1}}{2^p - 1}. \tag{4.37}$$

Zusätzlich kann das Richardson-Extrapolationsverfahren für die genauere Lösung einen lokalen Diskretisierungsfehler ϵ_κ abschätzen:

$$\epsilon_\kappa = \max_{i=1,\ldots,n} \frac{|y^h_{\kappa+1,i} - y^{h/2}_{\kappa+1,i}|}{(2^p - 1)\, S_i}, \tag{4.38}$$

mit den Skalierungsfaktoren

$$S_i = \text{atoli} + \text{rtoli} \cdot \max\left(|y_{\kappa,i}|, |y^{h/2}_{\kappa+1,i}|\right), \tag{4.39}$$

und den benutzerdefinierten Toleranzen atoli und rtoli.

4.12 Allgemeine lineare Verfahren

4.12.1 Grundlegendes

Die Oberklasse der *allgemeinen linearen Verfahren* (englisch: *General Linear Methods*, abgekürzt: *GLM*) wurden von der Arbeit [Z34] als Vereinheitlichung für Einschrittverfahren und

Mehrschrittverfahren entwickelt. Bei einem allgemeinen linearen Verfahren berechnet ein Zeitschritt κ ausgehend von den r *externen Stufenvektoren* $\mathbf{y}_1^{[\kappa]}, \ldots, \mathbf{y}_r^{[\kappa]}$ des vorherigen Zeitschritts $\kappa - 1$ mit Hilfe von s *internen Stufenvektoren* $\mathbf{Y}_1^{[\kappa]}, \ldots, \mathbf{Y}_s^{[\kappa]}$ seine r neuen externen Stufenvektoren $\mathbf{y}_1^{[\kappa+1]}, \ldots, \mathbf{y}_r^{[\kappa+1]}$ wie folgt:

$$\mathbf{Y}_i^{[\kappa]} = h \sum_{j=1}^{s} a_{ij} \mathbf{f}\left(\mathbf{Y}_j^{[\kappa]}\right) + \sum_{j=1}^{r} u_{ij} \mathbf{y}_j^{[\kappa]}, \quad i = 1, \ldots, s, \tag{4.40}$$

$$\mathbf{y}_i^{[\kappa+1]} = h \sum_{j=1}^{s} b_{ij} \mathbf{f}\left(\mathbf{Y}_j^{[\kappa]}\right) + \sum_{j=1}^{r} v_{ij} \mathbf{y}_j^{[\kappa]}, \quad i = 1, \ldots, r, \tag{4.41}$$

mit den Koeffizienten $A = (a_{ij}) \in \mathbb{R}^{s \times s}$, $B = (b_{ij}) \in \mathbb{R}^{r \times s}$, $U = (u_{ij}) \in \mathbb{R}^{s \times r}$ und $V = (v_{ij}) \in \mathbb{R}^{r \times r}$. Die Koeffizienten von einem allgemeinen linearen Verfahren lassen sich wiederum elegant in einem Butcher-Tableau zusammenfassen:

$$\begin{array}{c|c} A & U \\ \hline B & V \end{array} \quad . \tag{4.42}$$

Bei einem allgemeinen linearen Verfahren gilt auf Grund von Gleichung 4.41, dass ein Zeitschritt κ einen neuen externen Stufenvektor $\mathbf{y}_i^{[\kappa+1]}$ durch das Durchschleifen eines Stufenvektors $\mathbf{y}_m^{[\kappa]}$ mit $m \neq i$ aus dem vorherigen Zeitschritt $\kappa - 1$ erzeugen kann, so dass $\mathbf{y}_i^{[\kappa+1]} = \mathbf{y}_m^{[\kappa]}$ gilt. Dies ist wiederum bei folgenden Koeffizienten erfüllt:

$$b_{ij} = 0, \quad j = 1, \ldots, r, \tag{4.43}$$

$$v_{ij} = \begin{cases} 1 & \text{falls } j = m \\ 0 & \text{ansonsten} \end{cases} \quad . \tag{4.44}$$

Über diesen Umweg kann bei einem allgemeinen linearen Verfahren ein Zeitschritt κ effektiv für seine Berechnungen nicht nur die externen Stufenvektoren des Zeitschritts $\kappa - 1$, sondern auch die externen Stufenvektoren von den Zeitschritten $\kappa - 2, \ldots, \kappa - r$ verwenden. Dadurch eignet sich die Abstraktion der allgemeinen linearen Verfahren auch, um Mehrschrittverfahren zu beschreiben.

Des Weiteren kann bei einem allgemeinen linearen Verfahren ein interner oder ein externer Stufenvektor, trotz des Variablennamens $\mathbf{Y}$ beziehungsweise $\mathbf{y}$, einen Systemzustand oder eine zeitliche Ableitung des Systemzustands repräsentieren. Ein Stufenvektor, welcher eine zeitliche Ableitung repräsentiert, kann zum Beispiel erzeugt werden, indem seine Zeile in der Koeffizientenmatrix U (bei einem internen Stufenvektor) beziehungsweise V (bei einem externen Stufenvektor) nur Nulleinträge beinhaltet, oder auch, indem ihm über das zuvor beschriebene Durchschleifen eine zeitliche Ableitung aus einem vergangenen Zeitschritt zugewiesen wird.

Als Nächstes werden wir kurz erläutern, wie wir die RK-Verfahren und die Adams-Bashforth-Verfahren als allgemeines lineares Verfahren abstrahieren können.

4.12.2 RK-Verfahren als allgemeine lineare Verfahren

Um ein s-stufiges RK-Verfahren über ein allgemeines lineares Verfahren zu beschreiben, muss $s_{\text{GLM}} = s_{\text{RK}}$ sowie $r_{\text{GLM}} = 1$ gelten. Zusätzlich müssen die Koeffizienten des allgemeinen

linearen Verfahrens abhängig von Koeffizienten des RK-Verfahrens A_{RK} und $\mathbf{b}_{\text{RK}}$ wie folgt gewählt werden:

$$\begin{array}{c|c} A_{\text{GLM}} & U_{\text{GLM}} \\ \hline B_{\text{GLM}} & V_{\text{GLM}} \end{array} = \begin{array}{c|c} A_{\text{RK}} & (1, \dots, 1)^T \\ \hline \mathbf{b}_{\text{RK}} & 1 \end{array} . \tag{4.45}$$

Für diese Koeffizienten lässt sich die Formel für ein RK-Verfahren (Gleichungen 4.8, 4.9 sowie 4.10) leicht in die Formel des dazugehörigen allgemeinen linearen Verfahrens (Gleichungen 4.40 sowie 4.41) umformen.

4.12.3 Adams-Bashforth-Verfahren als allgemeine lineare Verfahren

Um ein k-Schritt Adams-Bashforth-Verfahren über ein allgemeines lineares Verfahren zu beschreiben, muss $s_{\text{GLM}} = 1$ sowie $r_{\text{GLM}} = k_{\text{AB}} + 1$ gewählt werden. Zusätzlich müssen die Koeffizienten des allgemeinen linearen Verfahrens abhängig von Koeffizienten des Adams-Bashforth-Verfahrens $\mathbf{b}_{\text{AB}}$ wie folgt festgelegt werden:

$$\begin{array}{c|c} A_{\text{GLM}} & U_{\text{GLM}} \\ \hline B_{\text{GLM}} & V_{\text{GLM}} \end{array} = \begin{array}{c|cccccl} 0 & 1 & b_1 & \dots & b_{k-1,\text{AB}} & b_{k,\text{AB}} & \text{(Nr. 1)} \\ \hline 0 & 1 & b_1 & \dots & b_{k-1,\text{AB}} & b_{k,\text{AB}} & \text{(Nr. 2)} \\ 1 & 0 & 0 & \dots & 0 & 0 & \text{(Nr. 3)} \\ 0 & 0 & 1 & \dots & 0 & 0 & \text{(Nr. 4)} \\ \vdots & \vdots & \vdots & \ddots & \vdots & 0 & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 & \text{(Nr. } k+2) \end{array} . \tag{4.46}$$

Mit dieser Butcher-Tableau-Belegung repräsentiert der interne Stufenvektor $\mathbf{Y}^{[\kappa]}_{1,\text{GLM}}$ den nächsten Systemzustand $\mathbf{y}_{\kappa+1,\text{AB}}$ im Adams-Bashforth-Verfahren. Dahingegen repräsentiert der erste externe Stufenvektor $\mathbf{y}^{[\kappa]}_{1,\text{GLM}}$ den aktuellen Systemzustand $\mathbf{y}_{\kappa,\text{AB}}$ im Adams-Bashforth-Verfahren. Zusätzlich repräsentieren die externen Stufenvektoren $\mathbf{y}^{[\kappa]}_{2,\text{GLM}}, \dots, \mathbf{y}^{[\kappa]}_{r,\text{GLM}}$ die zeitlichen Ableitungen $\mathbf{F}_{\kappa,\text{AB}}, \dots, \mathbf{F}_{\kappa-k-1,\text{AB}}$ der letzten k Zeitschritte im Adams-Bashforth-Verfahren, so dass $\mathbf{y}^{[\kappa]}_{2,\text{GLM}} = \mathbf{F}_{\kappa,\text{AB}}, \dots, \mathbf{y}^{[\kappa]}_{r,\text{GLM}} = \mathbf{F}_{\kappa-k-1,\text{AB}}$ gilt.

Dadurch beschreiben die 1. Zeile und 2. Zeile dieses Butcher-Tableaus jeweils die gleiche Berechnung von $\mathbf{y}_{\kappa+1,\text{AB}}$ im Adams-Bashforth-Verfahren (Gleichung 4.35). Dabei speichert die 1. Zeile ihr Ergebnis $\mathbf{y}_{\kappa+1,\text{AB}}$ in dem internen Stufenvektor $\mathbf{Y}^{[\kappa]}_{1,\text{GLM}}$ ab, während die zweite Zeile ihr Ergebnis in dem externen Stufenvektor $\mathbf{y}^{[\kappa+1]}_{1,\text{GLM}}$ abspeichert. Diesen internen Stufenvektor $\mathbf{Y}^{[\kappa]}_{1,\text{GLM}}$ verwendet die 3. Zeile dann um $\mathbf{F}_{\kappa+1,\text{AB}}$ des Adams-Bashforth-Verfahrens zu berechnen (Gleichung 4.36), welches sie in dem externen Stufenvektor $\mathbf{y}^{[\kappa+1]}_{2,\text{GLM}}$ abspeichert. Die darauffolgenden Zeilen 4 bis $k+2$ schleifen die externen Stufenvektoren $\mathbf{y}^{[\kappa]}_{3,\text{GLM}}, \dots, \mathbf{y}^{[\kappa]}_{r-1,\text{GLM}}$ mit den zeitlichen Ableitungen durch, so dass $\mathbf{y}^{[\kappa+1]}_{3,\text{GLM}} = \mathbf{y}^{[\kappa]}_{2,\text{GLM}} = \mathbf{F}_{\kappa,\text{AB}}, \dots, \mathbf{y}^{[\kappa+1]}_{r,\text{GLM}} = \mathbf{y}^{[\kappa]}_{r-1,\text{GLM}} = \mathbf{F}_{\kappa-k-2,\text{AB}}$ gilt. Dadurch beinhaltet jeder dieser externen Stufenvektoren wieder für den nächsten Zeitschritt $\kappa+1$ des Adams-Bashforth-Verfahrens die zeitliche Ableitung des passenden vergangenen Zeitschritts.

5 Klassen von Anfangswertproblemen und Auswahl repräsentativer Testprobleme

5.1 Klassifikation von Anfangswertproblemen

Es gibt eine sehr unüberschaubare Anzahl an wissenschaftlich oder ingenieurtechnisch relevanten Anfangswertproblemen, weshalb wir in dieser Arbeit nicht auf all diese eingehen können. Im folgenden Abschnitt wollen wir aber die Anfangswertprobleme nach unterschiedlichen Eigenschaften klassifizieren. Dabei fokussieren wir uns auf diejenigen Eigenschaften, welche für das Implementieren eines Lösers für Anfangswertprobleme durch ein ODE-Verfahren interessant erscheinen. Diese Klassifikation ist zudem ohne Anspruch auf Vollständigkeit. Allerdings hoffen wir, dass sie die wichtigsten Anfangswertprobleme und deren interessanteste Eigenschaften abdeckt. Die Eigenschaften, mit Hilfe derer wir Anfangswertprobleme klassifizieren können, lauten wie folgt:

Art der Simulation: Ausgewählte Beispiele für Simulationen mit zeitlich veränderliche Vorgängen, die somit das Lösen eines Anfangswertproblems erfordern, und die für konkrete Anwendungen zusätzlich noch oft miteinander gekoppelt werden müssen, sind:

- **Fluidsimulationen:** Diese simulieren das Fließverhalten einer Flüssigkeit, eines Gases oder eines Plasmas. Solche Fluidsimulationen werden zum Beispiel für die Wettervorhersage, die Katastrophenvorhersage sowie für viele ingenieurtechnische Anwendungen, wie für die Entwickelung von Dampfturbinen, Wasserkraftturbinen, Pumpen, Küstenschutz, Autos, Schiffen und Flugzeugen benötigt. Ebenso werden sie von der physikalischen Forschung benötigt, um zum Beispiel die Konvektion des Plasmas in einem Stern zu simulieren. Auch sind sie für biologische Vorgänge von Bedeutung, da sich mit ihnen zum Beispiel der Blutfluss in den Adern simulieren lässt.
- **Gravitative Simulationen:** Solche Simulationen berechnen die Bewegung von Körpern, die sich alle über die Gravitationskraft gegenseitig anziehen. Diese werden von der astrophysikalischen Forschung benötigt, um Fragen über die Vergangenheit und Zukunft des Universums oder zur Sternentstehung zu beantworten.
- **Chemische Simulationen:** Solche Simulationen berechnen, wie zwei oder mehrere chemische Substanzen über die Zeit miteinander reagieren. Da chemische Vorgänge meist räumlich inhomogen in einem flüssigen Medium stattfinden, werden chemische Simulationen oft mit einer Flüssigkeitssimulation oder oft auch nur einfacher mit einer Diffusionssimulation gekoppelt. Mögliche Anwendungen für solche Simulationen sind neben der chemischen Forschung die ingenieurtechnische Entwicklung von chemischen Reaktoren, Kolbenmotoren und Gasturbinen.
- **Simulation von verformbaren Körpern:** Solche Simulationen bestimmen, wie sich bewegende Körper unter Einwirkung von Kräften deformieren, indem sie zum Beispiel miteinander kollidieren oder sich selbst unter ihrer eigenen Last verformen. Die Simulation von verformbaren Körpern wird meist für ingenieurtechnische Simulationen, wie Unfallsimulationen, benötigt.
- **Simulation von Diffusion:** Diese simulieren die Diffusion einer Substanz in einem Fluid oder die Diffusion von Wärme in einem Festkörper oder einem Fluid. Die Simulation der

Diffusion ist oft als ein Teil einer anderen Simulation, zum Beispiel bei als ein Teil einer chemischen Simulation mit Diffusion, bedeutsam.

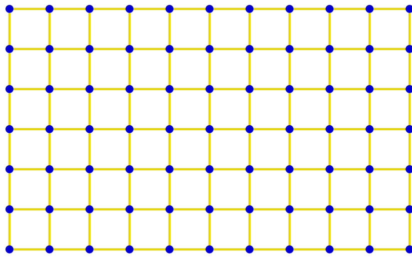
- **Simulation von Schaltkreisen:** Die Simulation von elektrischen und logischen Schaltkreisen wird zur Entwicklung neuer Computerhardware oder Elektrogeräten benötigt, während die Simulation von biologischen neuronalen Schaltkreisen von der biologischen Forschung oder der KI-Forschung benötigt wird.

Räumliche Struktur: Die mathematische Betrachtungsweise abstrahiert den Systemzustand eines Anfangswertproblems als einen einzigen Vektor. Bei vielen Anfangswertprobleme lassen sich jedoch jeweils eine oder mehrere Systemkomponenten zu einem Objekt gruppieren. Die so erzeugten Objekte sind oft wiederum in einer räumlichen Struktur angeordnet. Bei vielen Simulationen mit einer solchen räumlichen Struktur interagiert ein Objekt bevorzugt oder auch ausschließlich mit den anderen Objekten in seiner Nachbarschaft. Während diese Gruppierung von Systemkomponenten in Objekte und deren räumliche Struktur durch die mathematische Abstraktion verloren gehen, so ist diese Gruppierung jedoch gerade wegen ihrer Lokalität oft hilfreich, um die Auswertung der RHS-Funktion zu optimieren. Die wichtigsten räumlichen Strukturen von Anfangswertproblemen (siehe Abbildung 5.1) sind:

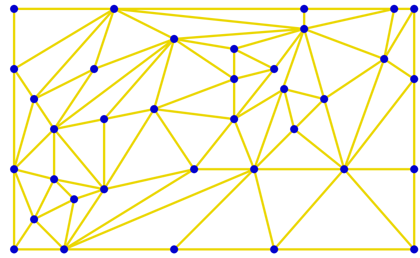
- **Gitter:** Bei solchen Anfangswertproblemen beschreibt der Systemzustand ein Gitter, wobei sich jeweils mehrere Komponenten des Systems zu einer Gitterzelle gruppieren lassen. Dieses Gitter kann eine unregelmäßige Topologie (unstrukturiertes Gitter) oder eine regelmäßige Topologie (strukturiertes Gitter) besitzen. Bei unstrukturierten Gittern muss die unregelmäßige Topologie in der Regel über eine zusätzliche Datenstruktur abgespeichert werden. Im Gegensatz dazu ist bei strukturierten Gittern keine solche Datenstruktur nötig. Bekannte Anfangswertprobleme mit einem unstrukturierten Gitter sind diejenigen Anfangswertprobleme, die sich über die Diskretisierung durch die Finite-Elemente-Methode ergeben (siehe zum Beispiel das Buch [B11] für eine Behandlung dieser Thematik). Anfangswertprobleme mit einem regelmäßigen Gitter sind zumeist Stencil-Probleme, welche sich aus der Diskretisierung von partiellen Differentialgleichungen durch die Linienmethode ergeben (siehe zum Beispiel das Buch [B12] für eine Behandlung dieser Thematik).

- **Graph mit räumlicher Struktur:** Bei solchen Anfangswertproblemen beschreibt der Systemzustand einen Graphen, wobei sich jeweils mehrere Komponenten des Systems zu einem Knoten des Graphen gruppieren lassen. Während die Knoten des Graphen zwar nicht direkt eine räumliche Position besitzen, so lassen sich die Knoten bei den meisten dieser Anfangswertprobleme so im Raum anordnen, dass ein jeder Knoten hauptsächlich mit den Knoten in seiner Nachbarschaft gekoppelt ist, und es nur wenige Kopplungen zwischen weit entfernten Knoten gibt. Hierunter fallen zum Beispiel Anfangswertprobleme, wie man sie für die Simulation eines elektrischen oder logischen Schaltkreis (siehe zum Beispiel [B13] für eine Behandlung dieser Thematik) lösen muss.

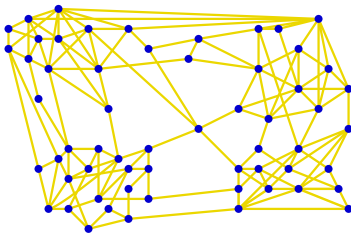
- **Bewegliche Partikel:** Bei solchen Anfangswertproblemen beschreibt der Systemzustand eine Menge von Partikeln, wobei sich jeweils mehrere Komponenten des Systems zu einem Partikel gruppieren lassen. Jedes dieser Partikel besitzt zumindest eine zeitlich veränderliche Position im Raum und eine Geschwindigkeit. Bei der Simulation bewegen sich die Partikel und interagieren dabei, das heißt, sie ziehen sich an oder stoßen sich ab. Diese Interaktion kann langreichweitig, wie zum Beispiel bei der Gravitationskraft beim



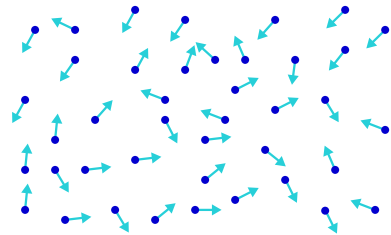
Strukturiertes Gitter



Unstrukturiertes Gitter



Graph mit räumlicher Struktur



Bewegliche Partikel

Abbildung 5.1: Klassifikation der räumlichen Strukturen von Anfangswertproblemen.

N-Körperproblem (siehe [B14]), oder auch nur kurzreichweitig, wie zum Beispiel die abstoßende Kraft zwischen Partikeln bei einer SPH-Flüssigkeitssimulation (siehe [Z35]), sein.

Ebenso sind Mischformen der soeben genannten räumlichen Strukturen möglich. So besteht zum Beispiel bei diversen Particle-in-Cell-Ansätzen der Systemzustand sowohl aus einem Gitter als auch aus einer Menge von Partikeln (siehe zum Beispiel [Z36]).

Systemparallelität der RHS-Funktion: Anfangswertprobleme lassen sich auch nach dem Grad der Systemparallelität ihrer RHS-Funktion klassifizieren. Für die Anfangswertprobleme, bei welchen sich jeweils mehrere Systemkomponenten zu einem Objekt, wie einem Partikel oder einer Gitterzelle, zusammenfassen lassen, gilt in der Regel, dass sich die einzelnen Objekte effizient parallel zueinander berechnen lassen, während sich die Systemkomponenten innerhalb eines Objekts auf Grund von Datenabhängigkeiten nur unter der Inkaufnahme von redundanten Berechnungen parallel zueinander berechnen lassen. Selbst wenn man mathematisch gesehen leicht RHS-Funktionen ohne Systemparallelität entwerfen kann, so gilt jedoch, dass für die meisten relevanten Anfangswertprobleme mit einer großen Systemgröße die RHS-Funktion auch einen hohen Grad an Systemparallelität besitzt.

Steifheit: Anfangswertprobleme lassen sich gemäß ihrer Steifheit klassifizieren (siehe zum Beispiel [B9] für eine ausführliche Behandlung dieser Thematik):

- **Nicht-steife Anfangswertprobleme:** Bei einem *nicht-steifen Anfangswertproblem* verursachen kleine Änderungen im Systemzustand immer auch nur kleine Änderungen in dessen zeitlicher Ableitung. Ein Beispiel für ein nicht-steifes Anfangswertproblem sind zwei schwach miteinander gekoppelte Pendel, bei welchem eine kleine Positionsänderung der beiden Pendel auch nur die Kraft der Kopplung zwischen den beiden Pendeln geringfügig ändert.
- **Steife Anfangswertprobleme:** Bei einem *steifen Anfangswertproblem* verursachen kleine Änderungen im Systemzustand potentiell sehr große Änderungen in dessen zeitlicher Ableitung. Ein Beispiel für ein steifes Anfangswertproblem ist der elastische Stoß zwischen zwei Kugeln, bei welchem eine kleine Positionsänderung der beiden Kugeln potentiell eine sehr große abstoßende Kraft zwischen diesen beiden Kugeln bewirkt.

Mathematisch lässt sich die Steifheit eines Anfangswertproblems über die Lipschitzbedingung definieren:

$$\|\mathbf{f}(t, \mathbf{y}_1) - \mathbf{f}(t, \mathbf{y}_2)\| \leq L\|\mathbf{y}_1 - \mathbf{y}_2\|, \tag{5.1}$$

wobei bei einem steifen Anfangswertproblem die Lipschitzkonstante $L \in \mathbb{R}^+$ sehr große Werte $L \gg 1$ annimmt. Dabei gilt, dass sich nicht-steife Anfangswertprobleme mit weniger Rechenaufwand durch ein explizites ODE-Verfahren lösen lassen, während sich steife Anfangswertprobleme mit weniger Rechenaufwand durch ein implizites ODE-Verfahren lösen lassen.

Anzahl an Pässen der RHS-Funktion: Anfangswertprobleme lassen sich danach klassifizieren, wie oft die RHS-Funktion über das System iterieren muss, das heißt wie viele *Pässe* die RHS-Funktion durchführen muss, um die zeitliche Ableitung des Systemzustands zu berechnen:

- **Single-Pass-Anfangswertprobleme:** Bei einem *Single-Pass-Anfangswertproblem* iteriert die RHS-Funktion nur einmal über alle Komponenten beziehungsweise über alle Objekte des ODE-Systems und berechnet dabei die zeitliche Ableitung für jede Komponente beziehungsweise für jedes Objekt.
- **Multi-Pass-Anfangswertprobleme:** Bei einem *Multi-Pass-Anfangswertproblem* muss die RHS-Funktion mehrmals über das gesamte System iterieren, um die zeitliche Ableitung des Systemzustands zu berechnen. Hierbei erzeugt jeder Pass eine Ausgabe, die wiederum als Eingabe für die darauffolgenden Pässe dient. Beispiele hierfür sind SPH-Flüssigkeitssimulationen, bei welchen die RHS-Funktion nacheinander mehrere Glättungskernel anwenden muss (siehe zum Beispiel [Z37]), oder High-Order-Stencils, die sich aus mehrmaliger Anwendung von Low-Order-Stencils, (siehe zum Beispiel [K13]) ergeben. Auch fallen diverse Partikel-In-Cell Ansätze in diese Kategorie, die in mehreren Pässen jeweils die Interaktionen zwischen den Partikeln und dem Gitter berechnen (siehe zum Beispiel [Z36]).

Bei manchen Multi-Pass-Anfangswertproblemen greifen aufeinanderfolgende Pässe relativ lokal auf die Eingabedaten zu, wodurch sich jeweils mehrere Pässe durch Blocking oder Tiling zu einem Pass fusionieren lassen. Deswegen ist die Unterscheidung zwischen einem Single-Pass-Problem und einem Multi-Pass-Problem oft nicht eindeutig.

Zeitlich veränderliche Anzahl an Systemkomponenten: Bei Anfangswertproblemen kann sich die Anzahl an Systemkomponenten mit der Simulationszeit ändern. Beispiele hierfür sind partikelbasierte Flüssigkeitssimulationen, bei welchen sich die Menge der simulierten Flüssigkeit und damit die Zahl der simulierten Partikel in der Simulationsdomäne durch einen Zufluss beziehungsweise einen Abfluss mit der Zeit ändert. Ein weiteres Beispiel hierfür sind adaptive Fluidsimulationen (siehe zum Beispiel [Z38]), bei welchen die Simulationsdomäne in Regionen unterteilt wird, wobei jede Region die Flüssigkeit in einer zeitlich veränderlichen Auflösung simuliert, wodurch sich auch die Anzahl der simulierten Systemkomponenten mit der Zeit ändert.

Zeitlich veränderliche Hilfsdatenstrukturen für die RHS-Funktion: Bei vielen Anfangswertproblemen benötigt eine effiziente Implementierung der RHS-Funktion eine zeitlich veränderliche Hilfsdatenstruktur, um die Rechenkomplexität der RHS-Funktion mit der Systemgröße n von meist $O(n^2)$ auf $O(n{\cdot}\log(n))$ zu reduzieren. Da diese Hilfsdatenstruktur sich mit der Zeit beziehungsweise mit dem Systemzustand ändert, muss sie bei jedem Aufruf der RHS-Funktion neu gebaut werden. Ein Beispiel hierfür sind die räumlichen Datenstrukturen für kurzreichweitige Partikelsimulationen, durch welche die RHS-Funktion für die Berechnung der zeitlichen Änderungen eines Partikels nicht mehr alle anderen Partikel, sondern nur noch die Partikel in dessen Nachbarschaft betrachten muss (siehe zum Beispiel [Z39]). Des Weiteren verwenden viele gitterbasierte Flüssigkeitssimulationen eine Datenstruktur für das Empty-Space-Skipping, mit Hilfe derer die RHS-Funktion Gitterbereiche überspringen kann, in denen sich keine Flüssigkeit befindet (siehe zum Beispiel [Z40]).

Zeitlich invariante Daten: Bei vielen Anfangswertproblemen besteht ein Systemzustand auch aus zeitlich invarianten Daten, auf welche die RHS-Funktion zugreifen muss. Hierunter fällt zum Beispiel eine zeitlich invariante Masse für jedes Partikel beim N-Körper-Problem oder bei der Simulation eines elektrischen Schaltkreises der Aufbau von ebendiesen.

Besetztheit des Anfangswertproblems: Die *Besetztheit* eines Anfangswertproblems gibt an, welche beziehungsweise wie viele Komponenten die RHS-Funktion für die Berechnung der zeitlichen Ableitung einer Komponente betrachten muss. Sie lässt sich dementsprechend noch einmal grob in dünnbesetzt und dichtbesetzt unterteilen (siehe zum Beispiel [K33]):

- **Dichtbesetzte Anfangswertprobleme:** Bei einem *dichtbesetzten Anfangswertproblem* muss die RHS-Funktion für die Berechnung der zeitlichen Änderung einer Komponente auf die meisten oder sogar alle anderen Systemkomponenten zugreifen. Solche Anfangswertprobleme besitzen dementsprechend mindestens eine Rechenkomplexität von $O(n^2)$ bezüglich der Systemgröße n.
- **Dünnbesetzte Anfangswertprobleme:** Bei einem *dünnbesetzten Anfangswertproblem* muss die RHS-Funktion für die Berechnung der zeitlichen Änderung einer Komponente nur auf wenige andere Systemkomponenten zugreifen. Dementsprechend besitzen solche Anfangswertprobleme meist eine Rechenkomplexität von $O(n)$ oder $O(n \cdot \log(n))$ bezüglich der Systemgröße n.

Diese Besetztheit kann nicht nur zeitlich konstant sein, sondern sie kann sich auch mit der Zeit ändern.

Arithmetische Intensität des Anfangswertproblems: Anfangswertprobleme lassen sich auch gemäß der *arithmetischen Intensität* ihrer RHS-Funktion klassifizieren. Bei *Anfangswertproblemen mit einer hohen arithmetischen Intensität* ist es möglich, die RHS-Funktion so zu implementieren, dass sie für jeden durchgeführten Speicherzugriff eine Vielzahl Berechnungen

durchführt. Dahingegen können bei *Anfangswertproblemen mit einer niedrigen arithmetischen Intensität* sämtliche Implementierungen der RHS-Funktion für jeden Speicherzugriff nur wenige Berechnungen durchführen. Generell gilt, dass dichtbesetzte Anfangswertprobleme meist eine hohe arithmetische Intensität besitzen. Dahingegen sind sowohl dünnbesetzte Anfangswertprobleme mit einer hohen arithmetischen Intensität, wie zum Beispiel High-Order-Stencils, als auch mit einer niedrigen arithmetischen Intensität, wie zum Beispiel Low-Order-Stencils, weit verbreitet.

Kopplung der Systemkomponenten: Anfangswertprobleme lassen sich auch dadurch klassifizieren, wie die Komponenten des ODE-Systems miteinander gekoppelt sind (siehe zum Beispiel [K30] oder [Z24]):

- **Beliebig gekoppelte Anfangswertprobleme (siehe Abbildung 5.2):** Bei solchen *beliebig gekoppelten Anfangswertproblemen*, welche auch als *allgemeine Anfangswertprobleme* bezeichnet werden, können die Gleichungen beliebig miteinander gekoppelt sein, das heißt die Auswertung $f_j(t, \mathbf{y})$ einer Komponente j der RHS-Funktion darf auf alle Komponenten $y_1, \ \ldots \ , y_n$ des Argumentvektors zugreifen. Dadurch muss der komplette Argumentvektor vorliegen, bevor auch nur eine Komponente der RHS-Funktion ausgewertet werden kann. Dadurch wirkt die RHS-Funktion wie eine globale Barriere, wodurch viele Lokalitätsoptimierungen verhindert werden.

- **Anfangswertprobleme mit beschränkter Zugriffsdistanz (siehe Abbildung 5.3):** Bei einem *Anfangswertproblem mit beschränkter Zugriffsdistanz* muss die RHS-Funktion für die Auswertung $f_j(t, \mathbf{y})$ einer Komponente j nur auf die Komponenten in $\mathbf{y}$ in der Umgebung von j zugreifen. Genauer gesagt ist die Zugriffsdistanz $d(\mathbf{f})$ der kleinste Wert d, so dass die Komponentenfunktion $f_j(t, \mathbf{y})$, $j = 1, \ldots, n$ nur auf die Teilmenge $y_{j-d}, \ \ldots \ , y_{j+d}$ der Komponenten des Argumentvektors $\mathbf{y}$ zugreift. Eine *beschränkte Zugriffsdistanz* von $\mathbf{f}$ liegt vor, falls $d(\mathbf{f}) \ll n$. Anfangswertprobleme mit beschränkter Zugriffsdistanz ergeben sich unter anderem aus Stencil-Problemen oder aus elektrischen Schaltkreisen mit einer Blockstruktur. Durch dieses Zugriffsmuster wirkt die RHS-Funktion nicht mehr als globale Barriere, das heißt, es muss nicht mehr der komplette Argumentvektor vorliegen, bevor auch nur eine Komponente der RHS-Funktion ausgewertet werden kann. Hierdurch werden wiederum eine Vielzahl von Lokalitätsoptimierungen möglich.

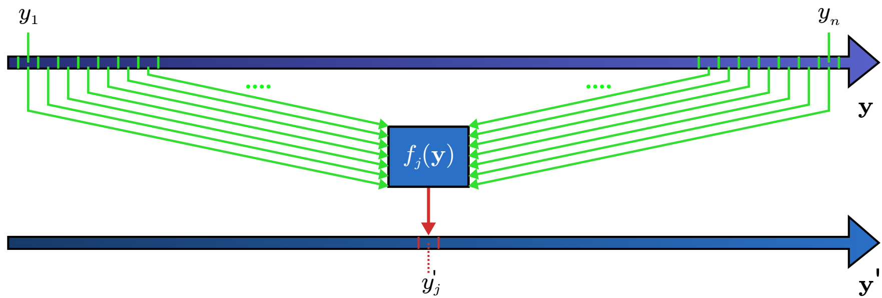


Abbildung 5.2: Zugriffsmuster einer RHS-Funktion für allgemeine Anfangswertprobleme. Bei allgemeinen Anfangswertproblemen ohne Einschränkung des Zugriffsmusters darf die RHS-Funktion für die Auswertung einer Komponente j auf alle Komponenten $y_1, \ldots, y_n$ im Argumentvektor zugreifen.

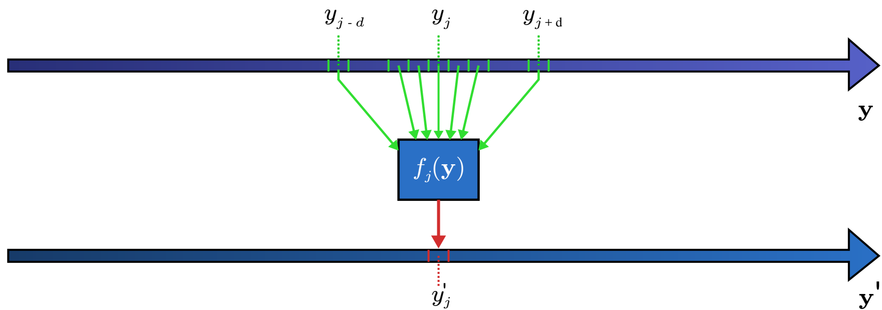


Abbildung 5.3: Zugriffsmuster einer RHS-Funktion für Anfangswertprobleme mit beschränkter Zugriffsdistanz. Bei Anfangswertproblemen mit beschränkter Zugriffsdistanz darf die RHS-Funktion für die Auswertung einer Komponente j nur auf die Komponenten $y_{j-d}, \ldots, y_{j+d}$ im Argumentvektor zugreifen.

5.2 Testproblem BRUSS2D

Unser Testproblem *BRUSS2D* beschreibt ein 2D-Reaktions-Diffusions-System basierend auf dem Brüsselator, welcher von der Arbeit [Z41] als ein Modell zur Beschreibung eines chemischen Oszillators entwickelt wurde. Bei BRUSS2D handelt es sich um ein dünnbesetztes Anfangswertproblem mit einer niedrigen arithmetischen Intensität. Auch lässt sich BRUSS2D als ein Stencil-Problem betrachten. Zudem ordnen wir für BRUSS2D die Komponenten des Brüsselators so im Zustandsvektor an, dass BRUSS2D eine beschränkte Zugriffsdistanz besitzt. BRUSS2D wird oft als Testproblem für ODE-Verfahren verwendet, um deren numerischen Eigenschaften (siehe zum Beispiel [B7]) und deren Performance (siehe zum Beispiel die Arbeiten [Z23], [K30] oder [Z24]) zu evaluieren.

Der Brüsselator beschreibt die oszillierende zeitliche Entwicklung der Konzentrationen zweier reagierender Substanzen $u(t)$ und $v(t)$ in einem flüssigen Medium. Dabei treten diese zwei Substanzen als Zwischenprodukte in einer anderen chemischen Reaktion auf, in welcher zwei Substanzen $a(t)$ und $b(t)$ miteinander in zwei andere Substanzen $c(t)$ und $d(t)$ reagieren:

$$\frac{da}{dt} = -k_1 a \quad , \tag{5.2}$$

$$\frac{db}{dt} = -k_2 bu \quad , \tag{5.3}$$

$$\frac{du}{dt} = +k_1 a \quad -k_2 bu \quad +k_3 u^2 v \quad -k_4 u, \tag{5.4}$$

$$\frac{dv}{dt} = +k_2 bu \quad -k_3 u^2 v \quad , \tag{5.5}$$

$$\frac{dc}{dt} = +k_2 bu \quad , \tag{5.6}$$

$$\frac{dd}{dt} = +k_4 u. \tag{5.7}$$

Dabei sind k_1, k_2, k_3, und k_4 die Reaktionsraten. Hierbei nehmen wir im Folgenden an, dass die Edukte a und b durch ein stetiges Nachführen konstant gehalten werden, während die Produkte c und d ständig abgeführt werden, wodurch wir nur noch $u(t)$ sowie $v(t)$ betrachten müssen.

Zusätzlich nehmen wir an, dass die chemische Reaktion in einem zweidimensionalen Flüssigkeitsfilm stattfindet, welcher durch die x- und y-Achse aufgespannt wird. In diesem Flüssigkeitsfilm sollen $u(t, x, y)$ und $v(t, x, y)$ nicht homogen verteilt sein, wodurch diese beiden Substanzen in dem Flüssigkeitsfilm diffundieren. Dadurch gilt für die Änderungsraten $\frac{du}{dt}$ und $\frac{dv}{dt}$ nun insgesamt:

$$\frac{\partial u}{\partial t} = +k_1 a \quad -k_2 bu \quad +k_3 u^2 v \quad -k_4 u \quad +D\Delta u, \tag{5.8}$$

$$\frac{\partial v}{\partial t} = +k_2 bu \quad -k_3 u^2 v \quad +D\Delta v. \tag{5.9}$$

Dabei ist D der Diffusionskoeffizient und Δ der Laplace-Operator.

Da diese beiden Gleichungen nun auf Grund ihrer zusätzlichen räumlichen Ableitungen keine gewöhnlichen, sondern partielle Differentialgleichungen sind, müssen wir sie nun per Ortsdis-

kretisierung in ein System aus gewöhnlichen Differentialgleichungen überführen. Diese Ortsdiskretisierung führen wir auf einem zweidimensionalen, kartesischen Gitter mit der Linienmethode durch, wobei wir als zweidimensionale Diskretisierung des Laplace-Operators den 5-Punkt-Stencil verwenden:

$$\frac{du_{x,y}}{dt} = +k_1 A \quad -k_2 B u_{x,y} \quad +k_3 u_{x,y}^2 v_{x,y} \quad -k_4 u_{x,y} \quad + D \cdot \frac{-4u_{x,y} + u_{x-1,y} + u_{x+1,y} + u_{x,y-1} + u_{x,y+1}}{l^2}, \tag{5.10}$$

$$\frac{dv_{x,y}}{dt} = +k_2 B u_{x,y} \quad -k_3 u_{x,y}^2 v_{x,y} \quad + D \cdot \frac{-4v_{x,y} + v_{x-1,y} + v_{x+1,y} + v_{x,y-1} + v_{x,y+1}}{l^2}. \tag{5.11}$$

Dabei beschreiben $u_{x,y}$ und $v_{x,y}$ die Konzentrationen der Substanzen u und v in der Gitterzelle mit den Koordinaten x, y. Zusätzlich ist l die Kantenlänge einer Gitterzelle. Das Gitter, auf welchem wir die chemische Reaktion simulieren, habe hierbei die Dimensionen von $x = 1, \ldots$, xmax sowie $y = 1, \ldots$, ymax. Am Rand des Gitters wenden wir die *Neumann-Randbedingungen* an:

$$\begin{aligned} u_{0,y} = u_{2,y},\ u_{\text{xmax}+1,y} = u_{\text{xmax}-1,y},\ u_{x,0} = u_{x,2},\ u_{x,\text{ymax}+1} = u_{x,\text{ymax}-1}, \\ v_{0,y} = u_{2,y},\ v_{\text{xmax}+1,y} = v_{\text{xmax}-1,y},\ v_{x,0} = u_{x,2},\ v_{x,\text{ymax}+1} = v_{x,\text{ymax}-1}. \end{aligned} \tag{5.12}$$

Die Gleichungen 5.10, 5.11 sowie 5.12 ergeben zusammen unser Testproblem BRUSS2D.

Wir legen das Gitter von BRUSS2D immer in der *Row-Major-Anordnung* im Vektor des Systemzustands $\mathbf{y}$ beziehungsweise im Speicher des Rechners ab, so dass sich zwei Gitterzellen, die entlang der x-Achse benachbart sind, sich auch benachbart im Vektor beziehungsweise benachbart im Speicher befinden. Des Weiteren gibt es mehrere unterschiedliche Möglichkeiten, wie man die Konzentrationen u und v der Gitterzellen im Speicher anordnen kann. Im Folgenden werden wir kurz zwei davon vorstellen:

- **Structure of Arrays:** In einer *Stucture of Arrays* (*SoA-Anordnung*) werden zuerst die u-Werte aller Gitterzellen nacheinander im Vektor beziehungsweise im Speicher abgelegt, danach folgen die v-Werte aller Gitterzellen:

$$\mathbf{y} = (u_{1,1}, \ldots, u_{\text{xmax,ymax}},\ v_{1,1}, \ldots, v_{\text{xmax,ymax}})^T . \tag{5.13}$$

- **Array of Structures:** In einem *Array of Structures* (*AoS-Anordnung*) werden jeweils der u- und der v-Wert einer Gitterzelle nacheinander im Vektor beziehungsweise im Speicher abgelegt, danach folgen der u- und der v-Wert der nächsten Gitterzelle:

$$\mathbf{y} = (u_{1,1}, v_{1,1}, \ldots, u_{\text{xmax,ymax}}, v_{\text{xmax,ymax}})^T . \tag{5.14}$$

Offensichtlich besitzt die AoS-Anordnung kombiniert mit der Row-Major-Anordnung eine beschränkte Zugriffsdistanz d von $d = 2 \cdot$ xmax, während die SoA-Anordnung keine solche beschränkte Zugriffsdistanz besitzt. Jedoch wäre es auch möglich, die Definition der beschränkten Zugriffsdistanz um ein SoA-Konzept zu erweitern, so dass diese SoA-Anordnung auch das

Kriterium der beschränkten Zugriffsdistanz erfüllen würde. Da wir in dieser Arbeit die beschränkte Zugriffsdistanz möglichst unkompliziert ausnutzen wollen, verwenden wir immer die AoS-Anordnung. Wir können zudem bei BRUSS2D eine beliebige Zugriffsdistanz d bei einer konstanten Problemgröße von n simulieren, indem wir für die x-Dimension des Gitters einen Wert von xmax $= d/2$ und für die y-Dimension des Gitters einen Wert von ymax $= n/(d/2)$ wählen.

Für eine effiziente Vektorisierung über aufeinanderfolgende Gitterzellen muss ein Prozessor die Vektorregister so mit Daten befüllen, dass ein Vektorregister entweder nur die u-Werte oder nur die v-Werte von aufeinanderfolgenden Gitterzellen beinhaltet. Auf GPUs ist dieses Befüllen sowohl mit der SoA- als auch AoS-Anordnung direkt über eine Gather-Instruktion mit perfektem Coalescing möglich (siehe zum Beispiel [T9]), wodurch beide Anordnungen ähnlich effizient sind. Dahingegen kann auf CPUs bei der SoA-Anordnung eine Gather-Instruktion oder eine SIMD-Block-Leseinstruktion die im Speicher alternierenden u- und v-Werte nicht direkt auf zwei Vektorregister aufteilen, wodurch das Laden in ein Vektorregister diese alternierende Anordnung von u- und v-Werten beibehält. Um nach dem Laden die u- und v-Werte jeweils in ihr eigenes Vektorregister zu kopieren, muss die CPU noch mehrere Shuffle- und Blend-Instruktionen ausführen (siehe zum Beispiel [T24]). Bei einer CPU-Implementierung von BRUSS2D mit der AoS-Anordnung muss diese komplizierte Vektorisierung des Ladens zusätzlich noch von dem CPU-Compiler erkannt werden. Scheitert der CPU-Compiler jedoch dies zu erkennen, so kann er die gesamte Schleifenstruktur von BRUSS2D nicht effizient vektorisieren. Dahingegen kann die CPU bei einer SoA-Anordnung die u- und v-Werte jeweils direkt über eine Gather-Instruktion oder SIMD-Block-Leseinstruktion in ihr eigenes Vektorregister laden. Dadurch ist diese SoA-Anordnung auf CPUs effizienter und ein CPU-Compiler kann sie auch leichter vektorisieren.

5.3 Testproblem NBODY

Unser Testproblem *NBODY*, welches auch als N-Körpersimulation bekannt ist, simuliert die Bewegung einer Menge von N punktförmigen Körpern (*Partikeln*), welche sich alle gegenseitig durch die Gravitationskraft anziehen. Dadurch handelt es sich bei unserem NBODY-Testproblem um ein dichtbesetztes Anfangswertproblem mit einer hohen arithmetischen Intensität. Eine ausführliche Behandlung von N-Körpersimulationen ist zum Beispiel in dem Buch [B14] zu finden.

Bei unserem Testproblem NBODY besitzt jedes Partikel i als Zustand eine zeitlich veränderliche Position $\mathbf{r}_i(t) \in \mathbb{R}^3$, Geschwindigkeit $\mathbf{v}_i(t) \in \mathbb{R}^3$ und eine zeitlich konstante Masse $m_i \in \mathbb{R}$. Somit werden jeweils sechs Systemkomponenten benötigt, um den zeitlich veränderlichen Zustand eines Partikels zu beschreiben, wodurch sich ein ODE-System bestehend aus $6 \cdot N$ Komponenten ergibt. Dementsprechend muss die RHS-Funktion beim NBODY-Problem für jedes Partikel i die zeitliche Änderung der Position $\frac{d\mathbf{r}_i}{dt}$ und der Geschwindigkeit $\frac{d\mathbf{v}_i}{dt}$ berechnen. Dabei gilt, dass die zeitliche Änderung der Position $\frac{d\mathbf{r}_i}{dt}$ eines Partikels i per Definition dessen Geschwindigkeit $\mathbf{v}_i$ ist:

$$\frac{d\mathbf{r}_i}{dt} = \mathbf{v}_i. \tag{5.15}$$

Deshalb muss eine Implementierung der RHS-Funktion diese Geschwindigkeit nur an diejenige Stelle im Ausgabevektor kopieren, an welcher sich die zeitliche Ableitung der Position des

Partikels befinden soll.

Die zeitliche Änderung der Geschwindigkeit $\frac{d\mathbf{v}_i}{dt}$ eines Partikels i ist per Definition dessen Beschleunigung $\mathbf{a}_i$, welche sich aus der Superposition $\sum \mathbf{F}$ aller Kräfte $\mathbf{F}$, die auf das Partikel einwirken, geteilt durch dessen Masse m_i ergibt. Da die Gravitation eine unbegrenzte Reichweite besitzt, wird auf das Partikel i von jedem anderen Partikel j eine Kraft $\mathbf{F}_{ij}$ ausgeübt, welche sich wiederum über das Gravitationsgesetz berechnet:

$$\frac{d\mathbf{v}_i}{dt} = \frac{1}{m_i} a_i \tag{5.16}$$

$$= \frac{1}{m_i} \sum_{\substack{j=1 \\ j \neq i}}^{N} \mathbf{F}_{ij} \tag{5.17}$$

$$= \frac{1}{m_i} \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{G m_i m_j \left(\mathbf{r}_j - \mathbf{r}_i\right)}{\left|\mathbf{r}_j - \mathbf{r}_i\right|^3} \tag{5.18}$$

$$= G \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{m_j \left(\mathbf{r}_j - \mathbf{r}_i\right)}{\left|\mathbf{r}_j - \mathbf{r}_i\right|^3}. \tag{5.19}$$

Hierbei ist $G \in \mathbb{R}$ die Gravitationskonstante. Somit muss eine Implementierung der RHS-Funktion, um die Beschleunigung eines Partikels zu berechnen, einmal über sämtliche anderen Partikel iterieren. Deshalb handelt es sich beim NBODY-Problem um ein dichtbesetztes Anfangswertproblem.

Für unsere Implementierung von dem Testproblem NBODY packen wir jeweils die Position $\mathbf{r}_i$ und die Masse m_i eines Partikels i in eine Variable rm4_i vom Typ `float4` beziehungsweise `double4`, so dass $\text{rm4}_i = (r_{i,x}, r_{i,y}, r_{i,z}, m_i)$ gilt. Ebenso speichern wir jeweils die Geschwindigkeit eines Partikels i in einer Variable v4_i vom Typ `float4` beziehungsweise `double4` ab, und lassen dabei die vierte Komponente ungenutzt, so dass $\text{v4}_i = (v_{i,x}, v_{i,y}, v_{i,z}, 0)$ gilt. Des Weiteren speichern wir im Zustandsvektor zuerst die Werte von rm4 für alle Partikel und danach die Werte von v4 für alle Partikel ab:

$$\mathbf{y} = \left(\text{rm4}_1, \ \ldots \ , \text{rm4}_N, \ \text{v4}_1, \ \ldots \ , \text{v4}_N\right)^T. \tag{5.20}$$

Diese Anordnung bewirkt, dass die Berechnung der Anziehungskräfte komplett auf dem ersten Teil des Vektors arbeiten kann, und sie die für eine Partikelinteraktion benötigten Daten effizient aus dem Speicher laden kann. Allerdings bleibt bei dieser Anordnung auch im Zustandsvektor pro Partikel eine Komponente ungenutzt, und eine andere Komponente wird dafür zweckentfremdet, zeitinvariante Daten abzuspeichern.

6 Datenflussgraphrepräsentation für ODE-Verfahren

6.1 Aufbau eines Datenflussgraphen

In dieser Arbeit verwenden wir eine *Datenflussgraphrepräsentation* für ODE-Verfahren, welche den Zeitschritt eines ODE-Verfahrens über einen Datenflussgraphen abstrahiert. Dabei beschreiben die Knoten im Datenflussgraphen die Berechnungen eines Zeitschritts des ODE-Verfahrens. Diese Knoten werden wir im Folgenden als Grundoperationen bezeichnen. Dadurch beschreiben die Kanten im Datenflussgraphen den Datenfluss zwischen diesen Grundoperationen. Eine solche Datenflussgraphrepräsentation wird beispielhaft für das Heun-Euler-Verfahren von der Abbildung 6.1 gezeigt. Für diese Arbeit klassifizieren wir die Grundoperationen im Datenflussgraphen eines ODE-Verfahrens wie folgt:

- **Vektorgrundoperationen:** Die *Vektorgrundoperationen* im Datenflussgraphen arbeiten auf Vektoren der Systemgröße wie Systemzuständen, zeitlichen Ableitungen und Fehlervektoren. Als Vektorgrundoperationen treten in ODE-Verfahren typischerweise Auswertungen der rechten Seite, Linearkombinationen, Map-Operationen und Reduktionen auf.
- **Konstantengenerierung:** Die *Konstantengenerierung* nimmt die aktuelle Zeit t_κ sowie die aktuelle Schrittweite h_κ entgegen und berechnet aus den *zeitinvarianten Koeffizienten* des ODE-Verfahrens die skalaren *zeitvarianten Konstanten* für die Vektorgrundoperationen, insbesondere die Gewichte für die Linearkombinationen. Bei vielen ODE-Verfahren sind die Berechnungen der Konstantengenerierung sehr einfach. So muss die Konstantengenerierung bei expliziten RK-Verfahren nur $h_\kappa \cdot A$, $h_\kappa \cdot \mathbf{b}$, $h_\kappa \cdot \hat{\mathbf{b}}$ und $t_\kappa + h_\kappa \cdot \mathbf{c}$ berechnen. Jedoch kann die Konstantengenerierung bei anderen ODE-Verfahren, wie zum Beispiel bei den Peer-Verfahren mit Schrittweitenkontrolle, auch sehr komplex sein.
- **Schrittweitensteuerung:** Die *Schrittweitensteuerung* nimmt den lokalen Fehler $\epsilon_{\kappa+1}$ entgegen, und bestimmt, ob der Zeitschritt angenommen oder verworfen werden soll. Für denjenigen Fall, dass der Zeitschritt angenommen werden soll, berechnet die Schrittweitensteuerung einen neue vergrößerte Schrittweite $h_{\kappa+1}$ für den nächsten Zeitschritt, und für denjenigen Fall, dass der Zeitschritt verworfen werden soll, eine verkleinerte Schrittweite h_κ für den aktuellen Zeitschritt. Damit ist die Schrittweitensteuerung die einzige Grundoperation, die auch den Kontrollfluss beeinflusst. Da die Schrittweitensteuerung in dem Datenflussgraphen eines ODE-Verfahrens maximal einmal auftritt, wird kein komplexer Kontrollflussgraph für die Beschreibung eines ODE-Verfahrens benötigt.

Im Folgenden wollen wir auf die typischen Vektorgrundoperationen eines ODE-Verfahrens näher eingehen. Dabei werden wir auch deren Zugriffsmuster auf die Argumentvektoren, das wir für die Lokalitätsoptimierungen in den folgenden Kapiteln berücksichtigen müssen, näher betrachten:

- **RHS-Operationen (Auswertung der rechten Seite):** Die RHS-Funktion $\mathbf{f}(t, \mathbf{y})$, $\mathbf{f} : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ einer *RHS-Operation* nimmt einen Systemzustand $\mathbf{y}$ als Argument und eine Zeit t entgegen, um daraus die zeitliche Ableitung des Systemzustands zu diesem Zeitpunkt zu berechnen. Das Zugriffsmuster einer RHS-Funktion auf den Argumentvektor $\mathbf{y}$ hängt von dem spezifischen Anfangswertproblem ab:

  - **Beliebig gekoppelte Anfangswertprobleme:** Bei solchen Anfangswertproblemen können die Komponenten des ODE-Systems beliebig miteinander gekoppelt sein. Folglich darf die RHS-Funktion für die Auswertung einer bestimmten Komponente f_j potentiell auf alle Komponenten $y_0, \ldots, y_n$ des Argumentvektors $\mathbf{y}$ zugreifen. Deshalb handelt es sich bei einer RHS-Funktion für ein beliebig gekoppeltes Anfangswertproblem auch um eine globale Barriere, welche Lokalitätsoptimierungen verhindert.
  - **Anfangswertprobleme mit beschränkter Zugriffsdistanz:** Bei solchen Anfangswertproblemen darf die RHS-Funktion für die Auswertung einer Komponente f_j nur auf die Komponenten $y_{j-d}, \ldots, y_{j+d}$ des Argumentvektors $\mathbf{y}$ innerhalb der beschränkten Zugriffsdistanz d zugreifen. Dieses Zugriffsmuster der beschränkten Zugriffsdistanzen werden wir später für das Tiling ausnutzen.

- **LC-Operationen (Linearkombinationen):** Bei ODE-Verfahren berechnen *LC-Operationen* über die Linearkombination von einem oder mehreren Systemzuständen oder zeitlichen Ableitungen einen neuen Systemzustand, eine zeitliche Ableitung oder einen Fehlervektor. Offensichtlich muss eine Linearkombination, und damit die dazugehörige LC-Operation, um die $j$te Komponente des Ergebnisvektors $\mathbf{y}$ zu berechnen, auf jeweils die $j$te Komponente von allen Argumentvektoren $\mathbf{x}_1, \ldots, \mathbf{x}_n$ zugreifen:

$$y_j = w_1 x_{1,j} + \ldots + w_n x_{n,j}, \quad j = 1, \ldots, n. \tag{6.1}$$

  Dabei sind $w_1, \ldots, w_m$ die Gewichte der Argumentvektoren der Linearkombination.

- **MAP-Operationen (Map-Funktionen):** Eine *MAP-Operation* beziehungsweise eine Map-Funktion $\mathbf{M}(g, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m)$ nimmt eine skalare Funktion $g : \mathbb{R}^m \to \mathbb{R}$ und einen oder mehrere Vektoren $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m \in \mathbb{R}^n$ als Argumente entgegnen. Für ihre Berechnung wendet die MAP-Operation dann die skalare Funktion g komponentenweise auf die Argumentvektoren an. Somit berechnet die MAP-Operation $\mathbf{M}$ ihre $j$te Komponente als:

$$M_j(g, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m) = g(x_{1,j}, x_{2,j}, \ldots, x_{m,j}), \quad j = 1, \ldots, n. \tag{6.2}$$

  Folglich greift die Berechnung der $j$ten Komponente von $\mathbf{M}$ nur auf jeweils die $j$te Komponenten aller Argumentvektoren $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ zu. Aus diesem Grund handelt es sich auch bei einer LC-Operation um eine besondere Art der MAP-Operation. ODE-Verfahren verwenden MAP-Operationen, die nicht auch LC-Operationen sind, zum Beispiel für die Berechnung des Fehlervektors (siehe zum Beispiel Gleichung 4.32 des Peer-Verfahrens). Ebenso benötigen Extrapolationsverfahren eine MAP-Operation für die Extrapolation der Lösung eines Makroschritts (siehe zum Beispiel Gleichung 4.37 des Richardsons-Extrapolationsverfahren).

- ***RED-Operationen*** **(Reduktionen):** Eine RED-Operation $R : \mathbb{R}^n \to \mathbb{R}$ reduziert einen Vektor der Systemgröße zu einem Skalar. Ein ODE-Verfahren benötigt für seine Schrittweitensteuerung typischerweise eine RED-Operation beziehungsweise eine Reduktion am Ende eines jeden Zeitschritts, um aus einem Fehlervektor $\mathbf{E}$ den lokalen Diskretisierungsfehler ϵ zu berechnen. Anschließend wird dieser lokale Diskretisierungsfehler dann von der Schrittweitensteuerung weiterverarbeitet. Für eine solche Reduktion wird meist die euklidische Norm oder die Maximumsnorm verwendet.

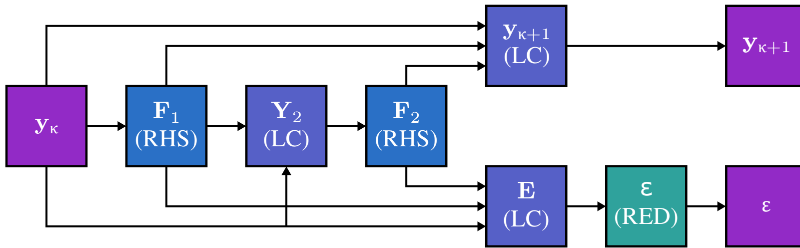


Abbildung 6.1: Datenflussgraph des Heun-Euler-Verfahrens. Das Heun-Euler-Verfahren ist ein häufig verwendetes, explizites, eingebettetes RK-Verfahren mit zwei Stufen, dessen Butcher-Tableau im Abschnitt 13.7 gezeigt wird.

Über einen Datenflussgraphen mit diesen obigen Grundoperationen können wir sämtliche Verfahren aus dem 4. Kapitel abdecken. Insbesondere können wir darüber die allgemeinen linearen Verfahren abdecken, da sich bei einem solchen allgemeinen linearen Verfahren sowohl die Berechnung der internen als auch der externen Stufen lediglich aus LC-Operationen und RHS-Operationen zusammensetzt (siehe Gleichungen 4.40 und 4.41). Da es sich bei den allgemeinen linearen Verfahren um eine Oberklasse handelt, können wir über unsere Datenflussgraphrepräsentation auch alle Unterklassen der allgemeinen linearen Verfahren abdecken. Hierunter fallen unter anderem die RK-Verfahren, Adams-Bashforth-Verfahren, Extrapolationsverfahren, PIRK-Verfahren und die Peer-Verfahren. So benötigt beispielsweise ein eingebettetes explizites RK-Verfahren für seine Stufenberechnung je eine LC-Operation und eine RHS-Operation pro Stufe, danach je eine weitere LC-Operation zur Bestimmung des nächsten Systemzustands, der eingebetteten Lösung und des Fehlervektors sowie abschließend eine RED-Operation zur Bestimmung des lokalen Diskretisierungsfehlers (Gleichungen 4.8 bis 4.14). Bei Adams-Bashforth-Verfahren besteht ein Zeitschritt aus einer RHS-Operation, gefolgt von einer LC-Operation (Gleichungen 4.35 und 4.36).

Wenn wir im Folgenden eine Aussage für mehrere Arten von Grundoperationen OP_1, OP_2, $\dots$, OP_n treffen wollen, so stellen wir für diese Aussage die Aufzählung dieser betreffenden Grundoperationen als $(\mathrm{OP}_1 \mid \mathrm{OP}_2 \mid \dots \mid \mathrm{OP}_n)$ dar. So heißt zum Beispiel eine Aussage bezüglich (RHS | LC | MAP)-Operationen, dass diese Aussage für RHS-Operationen, LC-Operationen und MAP-Operationen gültig ist. Eine ähnliche Nomenklatur verwenden wir für Abhängigkeiten. So heißt zum Beispiel eine Aussage bezüglich (RHS | LC) $\rightarrow$ (LC | MAP)-Abhängigkeiten, dass diese Aussage für RHS $\rightarrow$ LC-, RHS $\rightarrow$ MAP-, LC $\rightarrow$ LC- sowie LC $\rightarrow$ MAP-Abhängigkeiten gültig ist.

6.2 Abhängigkeiten zwischen Grundoperationen

Während der Datenflussgraph eines ODE-Verfahrens eine beliebige Struktur besitzen darf, so können in ihm nur bestimmte Abhängigkeiten zwischen den Grundoperationen auftreten. Diese möglichen Abhängigkeiten werden von Tabelle 6.1 gezeigt. Insbesondere gilt für die Abhängigkeiten Folgendes:

- RHS $\rightarrow$ RHS-Abhängigkeiten können nicht auftreten, da die RHS-Funktion einen Sys-

$a \to b$	b					
a	RHS	LC	MAP	RED	Konstgen.	Schrittstrg.
RHS	x	✓	✓	✓	x	x
LC	✓	✓	✓	✓	x	x
MAP	✓	✓	✓	✓	x	x
RED	x	x	x	x	x	✓
Konstgen.	✓	✓	✓	✓	✓	x
Schrittstrg.	x	x	x	x	✓	✓

Tabelle 6.1: Mögliche Abhängigkeiten in dem Datenflussgraphen eines ODE-Verfahrens.

temzustand entgegennimmt, um daraus dessen zeitliche Ableitung zu berechnen. Deshalb ergibt es mathematisch gesehen keinen Sinn, der RHS-Funktion eine zeitliche Ableitung als Argument zu übergeben, wie es im Falle einer RHS $\to$ RHS-Abhängigkeit geschehen würde.

- Eine Abhängigkeit von einem Konstantengenerierungsknoten zu einem anderen Konstantengenerierungsknoten ist erlaubt, damit die Konstantengenerierung über die Zeitschritte zusätzliche Informationen an sich selbst weitergeben kann. Dies wird nur bei manchen ODE-Verfahren, wie zum Beispiel bei den Peer-Verfahren mit Schrittweitensteuerung, benötigt.
- RED-Operationen können in unserem Ansatz ihr Ergebnis nur an die Schrittweitensteuerung weitergeben. Eine direkte Weitergabe des Ergebnisses einer RED-Operation an eine (RHS | LC | MAP)-Operation oder an eine Konstantengenerierung wäre aber prinzipiell auch realisierbar.

Zusätzlich gilt, dass bei (RHS | LC | MAP) $\to$ (LC | MAP)-Abhängigkeiten keine Abhängigkeiten zwischen den Komponenten des ODE-Systems auftreten, während bei (LC | MAP) $\to$ RHS-Abhängigkeiten sowie bei (RHS | LC | MAP) $\to$ RED-Abhängigkeiten ebendiese Abhängigkeiten zwischen den Komponenten des ODE-Systems auftreten. Da eine Implementierung einer RED-Operation aber die Reihenfolge der Reduktion unter Inkaufnahme eines sehr kleinen numerischen Fehlers beliebig umordnen darf, spielen diese Abhängigkeiten zwischen den Systemkomponenten bei (RHS | LC | MAP) $\to$ RED-Abhängigkeiten in unserer Arbeit keine Rolle.

6.3 Beschreibung von Abhängigkeiten über die Zeitschrittgrenzen

Um die Abhängigkeiten zwischen Grundoperationen über die Zeitschrittgrenzen hinweg zu modellieren, annotieren wir jede Abhängigkeit $a \to b$ zwischen zwei Grundoperationen a und b mit einer *Zeitschrittdistanz* $\Delta\kappa(a \to b) \in \mathbb{N}_0$ (siehe Abbildung 6.2 für die Zeitschrittdistanzen bei dem Midpoint-Verfahren und Abbildung 6.3 für die Zeitschrittdistanzen bei einem Adams-Bashforth-Verfahren). Falls die Zeitschrittdistanz 0 ist, dann sind sowohl die produzierende

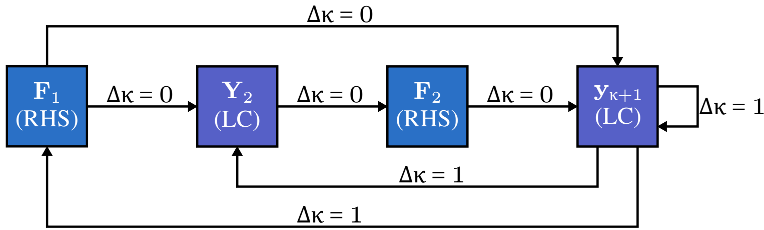


Abbildung 6.2: Datenflussgraph des Midpoint-Verfahrens mit annotierten Zeitschrittdistanzen. Bei dem Midpoint-Verfahren handelt es sich um ein weit verbreitetes explizites RK-Verfahren, dessen Butcher-Tableau im Abschnitt 13.7 gezeigt wird.

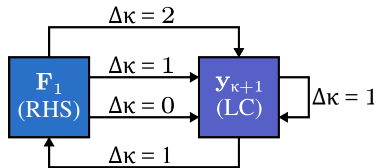


Abbildung 6.3: Datenflussgraph eines Adams-Bashforth-Verfahrens mit annotierten Zeitschrittdistanzen. In dieser Abbildung wird der Datenflussgraph desjenigen Adams-Bashforth-Verfahrens gezeigt, das für die Berechnung eines Zeitschritts die zeitlichen Ableitungen der letzten drei Zeitschritte benötigt, wodurch es sich um das Dreischritt-Adams-Bashforth-Verfahren handelt.

Operation a als auch die lesende Operation b Teil desselben Zeitschritts. Falls die Zeitschrittdistanz größer als 0 ist, dann verarbeitet die lesende Operation b das Ergebnis, welches die produzierende Operation a vor $\Delta\kappa(a \to b)$ Zeitschritten berechnet hat. Da konzeptbedingt keine Abhängigkeiten von einem zukünftigen Zeitschritt zu dem gegenwärtigen Zeitschritt existieren darf, muss $\Delta\kappa \geq 0$ sein. Zudem gilt, dass, wenn ein Algorithmus nur die Abhängigkeiten innerhalb eines Zeitschritts betrachten soll, er nur diejenigen Kanten mit $\Delta\kappa = 0$ betrachten darf, das heißt ein solcher Algorithmus muss alle übrigen Kanten mit $\Delta\kappa > 0$ ignorieren.

6.4 Datenflussgraphen der Verfahrensklassen

Als Nächstes wollen wir kurz auf die Eigenschaften der Datenflussgraphen der Verfahrensklassen eingehen. Zuerst betrachten wir die Datenflussgraphen von Einschritt- und Mehrschrittverfahren:

- **Einschrittverfahren:** Per Definition greifen Einschrittverfahren bei der Berechnung eines neuen Zeitschritts $\kappa + 1$ nur auf die Ergebnisse des vorherigen Zeitschritts κ zu. Aus diesem Grund können bei Einschrittverfahren nur Zeitschrittdistanzen $\Delta\kappa$ von 0 und 1 auftreten (siehe Abbildung 6.2 für die Zeitschrittdistanzen bei dem Midpoint-Verfahren).
- **Mehrschrittverfahren:** Per Definition greifen Mehrschrittverfahren bei der Berechnung eines neuen Zeitschritts $\kappa + 1$ nicht nur auf die Ergebnisse des letzten Zeitschritts κ,

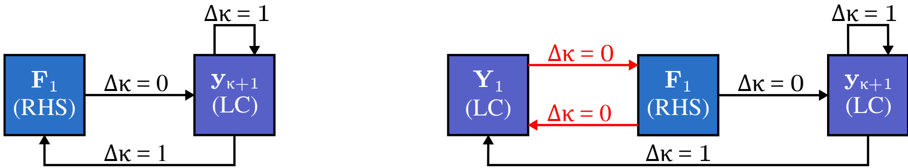


Abbildung 6.4: Die Datenflussgraphen vom expliziten und impliziten Euler-Verfahren: Diese Abbildung zeigt den Datenflussgraphen eines Zeitschritts des expliziten Euler-Verfahrens (links) und des impliziten Euler-Verfahrens (rechts), wobei die zyklischen Abhängigkeiten des impliziten Verfahrens rot markiert sind.

sondern auch auf die Ergebnisse der vorherigen Zeitschritte $\kappa - 1, \kappa - 2, \ \ldots$ zu. Aus diesem Grund tritt bei einem k-Schrittverfahren mindestens eine Zeitschrittdistanz $\Delta\kappa$ der Größe k auf (siehe Abbildung 6.3 für die Zeitschrittdistanzen bei einem Adams-Bashforth-Verfahren).

Die Datenflussgraphen von impliziten und expliziten ODE-Verfahren unterschieden sich wie folgt:

- **Explizite ODE-Verfahren:** Bei einem expliziten ODE-Verfahren existiert keine zyklischen Abhängigkeiten innerhalb seiner mathematischen Definition. Deshalb existieren im Datenflussgraphen eines solchen expliziten ODE-Verfahrens ebenfalls keine zyklischen Abhängigkeiten. Aus diesem Grund können wir leicht und automatisch aus dem Datenflussgraphen eines expliziten ODE-Verfahrens eine Basisimplementierung erstellen, welche für jeden Zeitschritt die Grundoperationen gemäß ihrer topologischen Reihenfolge nacheinander berechnet (siehe 8. Kapitel).
- **Implizite ODE-Verfahren:** Bei einem impliziten ODE-Verfahren existiert mindestens eine zyklische Abhängigkeit in seiner mathematischen Definition. Aus diesem Grund existiert im Datenflussgraphen eines impliziten ODE-Verfahrens ebenfalls mindestens eine zyklische Abhängigkeit innerhalb eines Zeitschritts (siehe Abbildung 6.4 für den Datenflussgraphen des impliziten Euler-Verfahrens). Für ein implizites ODE-Verfahren kann man den Datenflussgraphen ebenfalls dafür verwenden, um automatisch Implementierungsvarianten zu generieren. Eine solche Implementierungsvariante muss jedoch die Zyklen im Datenflussgraphen erkennen, zu einem nicht-linearen Gleichungssystem transformieren, und dieses dann mit einem entsprechenden Lösungsverfahren für solche nicht-linearen Gleichungssysteme, wie zum Beispiel dem Newton-Verfahren, lösen.

6.5 Einfache mathematische Optimierungen auf dem Datenflussgraphen

Als Nächstes werden wir kurz einfache mathematische Optimierungen auf den Datenflussgraphen von ODE-Verfahren betrachten, die uns in den folgenden Kapiteln helfen werden Berechnungen oder Speicherzugriffe einzusparen:

- Falls in der mathematischen Beschreibung eines ODE-Verfahrens eine Linearkombination Argumentvektoren mit dem Gewicht von null beinhaltet, können wir diese Vektoren aus der entsprechenden LC-Operation entfernen.
- Falls in der mathematischen Beschreibung eines ODE-Verfahrens eine Linearkombination nur einen Argumentvektor mit einem Gewicht von Eins besitzt, das heißt die Linearkombination eine neutrale Operation auf ihren einzigen Argumentvektor durchführt, dann müssen wir für diese Linearkombination keine LC-Operation in dem Datenflussgraphen einfügen und können stattdessen diesen Argumentvektor direkt an alle Grundoperationen, die das Ergebnis dieser neutralen Linearkombination benötigen, weiterleiten.
- Falls die mathematische Beschreibung eines ODE-Verfahrens redundante Berechnungen beinhaltet, zum Beispiel wenn in einem RK-Verfahren mehrere Zeilen im Butcher-Tableau mit Nullen gefüllt sind, können wir diese redundanten Berechnungen zu einer einzigen Grundoperation verschmelzen.
- Bei einer LC $\rightarrow$ LC-Abhängigkeit können wir die Argumentvektoren der ersten LC-Operation mit den Gewichten der ersten LC-Operation direkt zu den Argumentvektoren der davon abhängigen zweiten LC-Operation hinzuaddieren, wodurch wir diese Abhängigkeit zwischen den beiden LC-Operationen eliminieren.

6.6 Abhängigkeitskette als eine Struktur im Datenflussgraphen

Bei vielen expliziten ODE-Verfahren, wie zum Beispiel den RK-Verfahren, enthält die *transitive Reduktion* des Datenflussgraphen, also das Entfernen aller Kanten, die die topologische Sortierung nicht beeinflussen, eine *Abhängigkeitskette* aus *RHS* $\rightarrow$ *LC-Gliedern* mit einer optionalen RED-Operation am Ende der Abhängigkeitskette für die Schrittweitensteuerung (siehe Abbildung 6.5). Die LC-Operationen in dieser Abhängigkeitskette benötigen nicht nur die direkt vorhergehende RHS-Operation als Argument, sondern auch weitere zuvor berechnete RHS-Operationen und LC-Operationen. Innerhalb eines RHS $\rightarrow$ LC-Glieds der Abhängigkeitskette gibt es keine Abhängigkeiten zwischen unterschiedlichen Systemkomponenten, während zwischen unterschiedlichen RHS $\rightarrow$ LC-Gliedern ebendiese Abhängigkeiten zwischen unterschiedlichen Systemkomponenten vorhanden sind (siehe Abbildung 6.6).

Bei manchen ODE-Verfahren können zwei oder mehr RHS-Operationen oder LC-Operationen voneinander unabhängig sein. Hierunter fallen zum Beispiel RK-Verfahren mit unabhängigen Stufen, die PIRK-Verfahren oder die Peer-Verfahren (siehe Abbildung 6.7 oder Abbildung 6.8 für ein Beispiel). Ohne Beschränkung der Allgemeinheit können wir diese voneinander unabhängigen Vektorgrundoperationen zu einem Glied in der Abhängigkeitskette zusammenfassen, wodurch dann ein RHS $\rightarrow$ LC-Glied in der Abhängigkeitskette aus mehreren RHS-Operationen und LC-Operationen besteht.

Um diese Abhängigkeitskette zu konstruieren, können wir folgenden Algorithmus verwenden: Zunächst müssen wir für jede Vektorgrundoperation sämtliche Pfade suchen, die von einer Vektorgrundoperation am Anfang des Zeitschritts aus zu dieser Vektorgrundoperation führen. Für jeden dieser Pfade zählen wir die Anzahl an RHS-Operationen, die dieser Pfad beinhaltet. Diejenige maximale Anzahl an RHS-Operationen unter allen Pfaden entspricht der Nummer des RHS $\rightarrow$ LC-Glieds innerhalb der Abhängigkeitskette, in welches wir die Vektorgrundoperation einfügen müssen.

Alternativ könnten wir auch die Abhängigkeitskette so definieren, dass sie aus LC $\rightarrow$ RHS-Gliedern besteht. Vorteilhafterweise würde bei einer solchen LC $\rightarrow$ RHS-Abhängigkeitskette ein Glied nun einer Stufe eines Einschrittverfahrens, wie zum Beispiel der Stufe eines RK-Verfahrens, entsprechen. Jedoch wären bei einer solchen LC $\rightarrow$ RHS-Abhängigkeitskette auch die Kernelfusion und das Tiling in den folgenden Kapiteln deutlich umständlicher, da beide auf RHS $\rightarrow$ LC-Gliedern arbeiten.

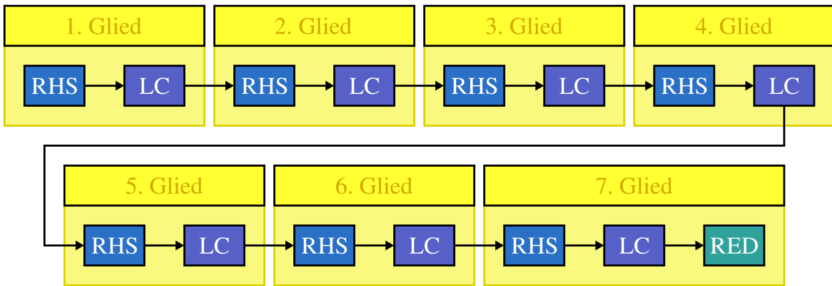


Abbildung 6.5: RHS → LC-Abhängigkeitskette aus RHS → LC-Gliedern. Diese Abbildung zeigt eine Abhängigkeitskette aus RHS → LC-Gliedern, die ein Teil der transitiven Reduktion des Datenflussgraphen von vielen Einschritt- oder Mehrschrittverfahren ist. Dabei verwenden die LC-Operationen in dieser Abhängigkeitskette auch weitere vorherige LC-Operationen oder RHS-Operationen als Argument. Diese Abhängigkeiten werden jedoch durch die transitive Reduktion entfernt.

Abbildung 6.6: Die Abhängigkeiten zwischen den Systemkomponenten in einer RHS → LC-Abhängigkeitskette. Diese Abbildung zeigt, dass bei RHS → LC-Abhängigkeiten keine Abhängigkeiten zwischen den Systemkomponenten auftreten, während eben solche Abhängigkeiten zwischen den Systemkomponenten bei LC → RHS-Abhängigkeiten vorhanden sind.

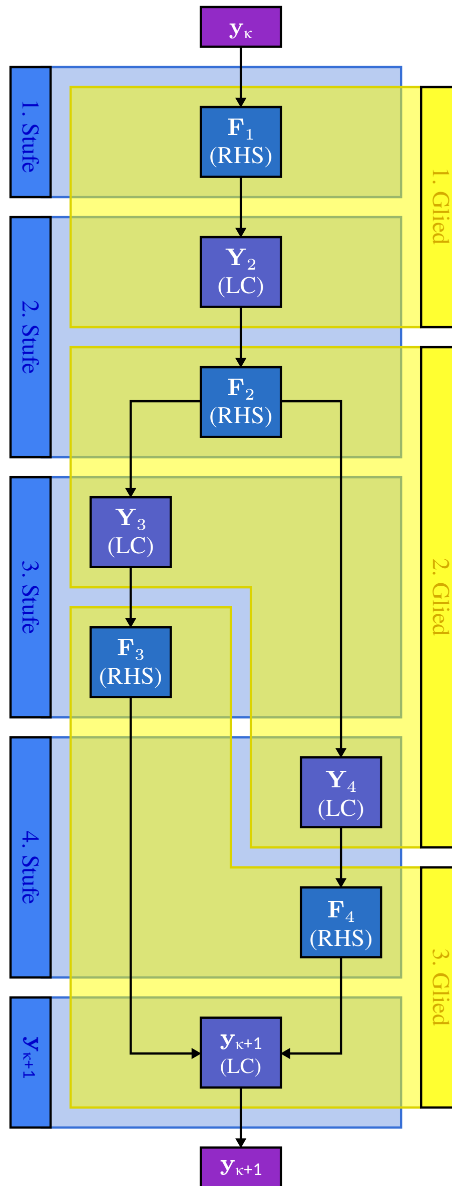


Abbildung 6.7: Datenflussgraph eines vierstufigen RK-Verfahrens mit zwei unabhängigen Stufen. Diese Abbildung zeigt die Beziehung zwischen den Stufen und der LC $\rightarrow$ RHS-Abhängigkeitskette für ein vierstufigen RK-Verfahren, bei welchem die dritte und die vierte Stufe unabhängig voneinander sind. Dabei zeigt die Abbildung nur diejenigen Abhängigkeiten, die nach einer transitiven Reduktion übrig bleiben.

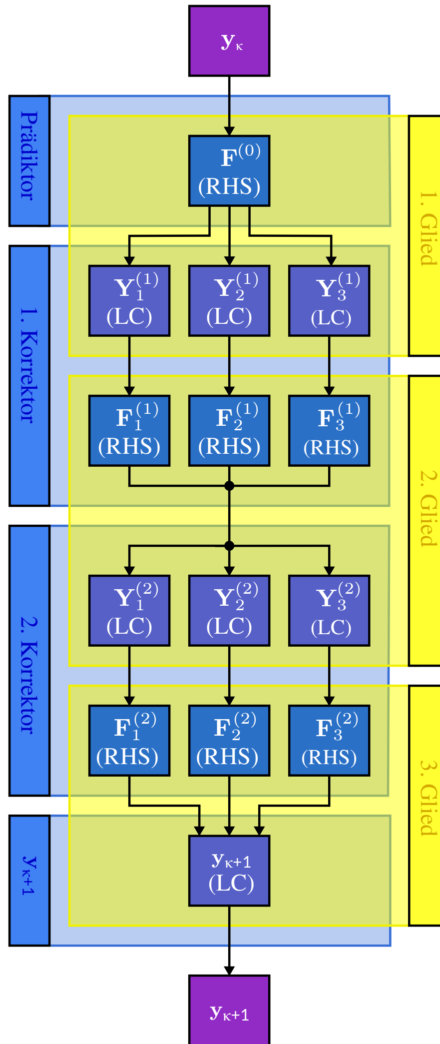


Abbildung 6.8: Datenflussgraph eines PIRK-Verfahrens. Die Abbildung zeigt den Datenflussgraphen eines PIRK-Verfahrens, das ein implizites RK-Verfahren mit drei Stufen als Basisverfahren verwendet, und die daraus resultierende Abhängigkeitskette. Zudem zeigt die Abbildung wieder nur diejenigen Abhängigkeiten, die nach einer transitiven Reduktion übrig bleiben.

7 Grundkonzepte unserer Implementierungen

7.1 Motivation

Mit diesem kurzen Einschubkapitel werden wir einige sehr wichtige Grundkonzepte, also die Grundkonzepte der abstrakten Kernels, der permanenten Vektoren und der ausführbaren Objekte, hervorheben und vorstellen. Diese Grundkonzepte werden wir in dieser Arbeit über sämtliche Implementierungsvarianten, also der Basisimplementierung und den Implementierungsvarianten mit Kernelfusion und Tiling, hinweg verwenden.

7.2 Abstrakte Kernels

Für unsere Implementierungen definieren wir zunächst das Konzept des *abstrakten Kernels*. Dies ist eine parallele Subroutine, die eine Aufgabe bestehend aus parallelen *Blöcken* abarbeitet. In einem abstrakten Kernel lässt sich jeder Block weiterhin in eine Vielzahl von Rechenoperationen unterteilen, die untereinander einen hohen Grad an Parallelität und oft sogar an Lokalität besitzen. Beides kann die Implementierung eines abstrakten Kernels für die bessere Auslastung ihrer Rechenwerke ausnutzen. Für seine Abarbeitung weist ein abstraktes Kernel seine parallelen Blöcke den Kernen des Rechners zu. Da es bei einem abstrakten Kernel in der Regel mehr parallele Blöcke gibt, als der Rechner gleichzeitig bearbeiten kann, dürfen zwischen den parallelen Blöcken des Kernels keine Abhängigkeiten bestehen, da dies ansonsten zu einem Deadlock in dem abstrakten Kernel führen würde. Die Zuweisung zwischen parallelen Blöcken und Rechenkernen kann bei einem abstrakten Kernel per $n : 1$ oder auch per $n : m$ Mapping geschehen. Somit wird bei einem $n : 1$ Mapping jeder Block jeweils nur von einem Rechenkern abgearbeitet. Dahingegen kooperieren bei einem $n : m$ Mapping an einem Block beziehungsweise an einer Aufgabe jeweils m Rechenkerne, wodurch eine zusätzliche Synchronisation zwischen den Rechenkernen erforderlich ist. Bei der Basisimplementierung berechnet ein abstraktes Kernel zunächst nur jeweils eine Vektorgrundoperation. Bei den späteren Varianten mit Fusion oder Tiling berechnet ein abstraktes Kernel jedoch jeweils mehrere Vektorgrundoperationen.

Für unsere GPU-Implementierungen bilden wir im Folgenden immer jedes abstrakte Kernel auf jeweils ein GPU-Kernel ab. Falls das abstrakte Kernel ein $n : 1$ Mapping zwischen parallelen Blöcken und Rechenkernen besitzt, lassen wir jeweils einen parallelen Block durch eine Workgroup abarbeiten, wohingegen wir bei einem $n : m$ Mapping einen parallelen Block durch m Workgroups abarbeiten lassen.

Für CPU-Implementierungen bilden wir ein abstraktes Kernel auf ein paralleles Schleifennest ab, das wir als CPU-Kernel bezeichnen. Die äußerste Schleife in diesem CPU-Kernel ist immer eine parallele Schleife, die über die Blöcke des Kernels iteriert. Die Zuteilung der Blöcke auf die CPU-Kerne kann dynamisch oder statisch erfolgen. In unserer Implementierung verwenden wir immer eine dynamische Zuweisung über das Inkrementieren eines atomaren Zählers.

7.3 Permanente Vektoren

Wir bezeichnen im Folgenden einen Vektor der Systemgröße, mit Hilfe dem wir das Ergebnis einer (RHS | LC | MAP)-Operation zwischen zwei Kernels transferieren, als einen *permanenten*

Vektor. In der Basisimplementierung muss jede (RHS | LC | MAP)-Operation ihr Ergebnis immer in einem solchen permanenten Vektor abspeichern. Bei den späteren Varianten mit Fusion oder mit Tiling müssen nicht mehr die Ergebnisse von allen (RHS | LC | MAP)-Operationen zwischen den Kernels transferiert werden. Falls bei diesen Varianten ein solcher Transfer nicht mehr nötig ist, muss eine (RHS | LC | MAP)-Operation ihr Ergebnis auch nicht mehr in einem solchen permanenten Vektor abspeichern. Da wir in dieser Arbeit primär Systemgrößen betrachten, welche deutlich größer als der On-Chip-Speicher des Prozessors sind, findet der Datentransfer zwischen zwei Kernels durch einen permanenten Vektor auch nicht über den On-Chip-Speicher des Prozessors, sondern über dessen DRAM statt.

Deswegen verwenden wir in der folgenden Arbeit für unsere Implementierungen für x64-CPUs, auf welchen wir auch unsere experimentellen Untersuchungen durchführen werden, für das Schreiben in permanente Vektoren die Non-Temporal-Store-Instruktionen, um so den Fetch-On-Write-Overhead des Last-Level-Caches der x64-CPU zu vermeiden. Um diese Non-Temporal-Store-Instruktionen in den Pseudocode für unsere CPU-Implementierungen in den folgenden Kapiteln einzubauen, verwenden wir im Folgenden den <− Operator. Dahingegen verwenden wir für reguläre Store-Instruktionen, das heißt Store-Instruktionen, die von den Caches über die Write-Allocate- und Fetch-On-Write-Policy abgewickelt werden, den = Operator. Letztlich gilt anzumerken, dass die Non-Temporal-Store-Instruktionen nur bei großen Problemgrößen lohnenswert sind, für welche die CPU nur wenig Daten zwischen zwei aufeinanderfolgenden Kernels über ihre Caches wiederverwenden kann. Dahingegen sind Non-Temporal-Store-Instruktionen bei kleinen Problemgrößen kontraproduktiv, da sie diejenigen Daten, die die CPU potentiell über ihre Caches transferieren könnte, in den DRAM schreiben, so dass die CPU sie von dort erst wieder in ihre Caches laden muss. Dementsprechend sind bei kleinen Problemgrößen reguläre Store-Instruktionen besser geeignet, um Vektorkomponenten in permanente Vektoren zurückzuschreiben.

7.4 Ausführbare Objekte

In unserem Ansatz sind nur die Kernel, die Schrittweitenkontrolle und die Konstantengenerierung direkt ausführbar. Deshalb abstrahieren wir jeweils ein Kernel, eine Schrittweitenkontrolle oder eine Konstantengenerierung als ein *ausführbares Objekt*. Dahingegen sind die Vektorgrundoperationen nicht oder nur indirekt ausführbar, indem wir sie über ein Kernel berechnen lassen.

8 Basisimplementierung

8.1 Motivation

Da es für das Lösen von Anfangswertproblemen sehr aufwendig ist, für jede Kombination von ODE-Verfahren, Anfangswertproblem und Zielarchitektur von Hand eine optimierte Implementierungsvariante zu generieren, ist eine automatische Generierung von optimierten Implementierungsvarianten von ODE-Verfahren zum Lösen von Anfangswertproblemen vorteilhaft. Als einen Ausgangspunkt für die Lokalitätsoptimierungen in den folgenden Kapiteln stellen wir zuerst eine Basisimplementierung vor, die sich sehr stark an der Datenflussgraphrepräsentation von ODE-Verfahren orientiert. Dadurch lässt sich die Basisimplementierung für ein ODE-Verfahren auch leicht aus dessen Datenflussgraphen erzeugen.

8.2 Theorie

Unsere Basisimplementierung führt für jeden Zeitschritt die Grundoperationen im Datenflussgraphen nacheinander gemäß ihrer topologischen Ordnung aus. Dabei gilt für die Ausführung der verschiedenen Grundoperationen Folgendes:

- **Konstantengenerierung und Schrittweitenkontrolle:** Da die Konstantengenerierung und die Schrittweitenkontrolle im Gegensatz zu den Vektorgrundoperationen nur wenig Speicherzugriffe sowie Berechnungen durchführen, und zudem nur schlecht parallelisierbar sind, bieten sie kaum Potential für Optimierungen. Aus diesem Grund ist eine Parallelisierung oder eine Auslagerung dieser beiden Grundoperationen auf parallele Koprozessoren nur wenig sinnvoll, weshalb sie in unserer Arbeit nur sequentiell von der CPU berechnet werden. Wegen ihres geringen Anteils an der Gesamtlaufzeit vernachlässigen wir diese beiden Grundoperationen in der folgenden Arbeit weitestgehend.
- **Vektorgrundoperationen:** Da die Vektorgrundoperationen einen absoluten Großteil aller Berechnungen eines ODE-Verfahrens durchführen, dabei viele Daten aus dem DRAM laden sowie viele Daten in den DRAM schreiben, und zudem einen hohen Grad an Systemparallelität besitzen, bietet es sich an, diese nicht nur von der CPU, sondern auch von parallelen Koprozessoren wie GPUs oder FPGAs berechnen zu lassen. Innerhalb einer Vektorgrundoperation kann die Basisimplementierung die Systemparallelität ausnutzen, um alle Kerne beziehungsweise alle Rechenwerke des Zielrechners auszulasten. Deshalb liegt es für die Basisimplementierung nahe, jede Vektoroperation durch ein abstraktes Kernel zu implementieren. Somit erzeugt die Basisimplementierung für jede RHS-Operationen ein *RHS-Kernel*, für jede LC-Operationen ein *LC-Kernel*, für jede MAP-Operationen ein *MAP-Kernel* und für jede RED-Operationen ein *RED-Kernel*. Dabei kann die Basisimplementierung auf einem konkreten Zielsystem Implementierungen für diese Kernels verwenden, die speziell für die Architektur des Zielsystems optimiert sind. Falls ein Zielsystem einen zusätzlichen Grad von Parallelität benötigt, kann die Basisimplementierung prinzipiell die Kernels von voneinander unabhängigen Vektorgrundoperationen vom Zielsystem parallel zueinander berechnen lassen, um so die Methodenparallelität des ODE-Verfahrens auszunutzen. Da wir in dieser Arbeit nur große Systeme mit genügend Systemparallelität betrachten, integrierten wir diese einfache Optimierung nicht mehr in unsere Basisimplementierung.

8.3 Basisimplementierung auf GPUs

8.3.1 Implementierung

Wenn unsere Basisimplementierung eine GPU als Zielarchitektur verwenden soll, so startet sie für jedes abstrakte Kernel beziehungsweise für jede Vektorgrundoperation ein spezialisiertes *GPU-Kernel*. Dabei besitzen diese GPU-Kernels, deren schematischer Pseudocode in Abbildung 8.1 gezeigt wird, folgende Struktur:

- **RHS-Kernel:** Leider ist es nicht möglich, allgemein ein effizientes RHS-Kernel für ein beliebiges Anfangswertproblem mit einer beliebigen RHS-Funktion für GPUs zu implementieren. Allerdings implementieren wir zwei einfache Optimierungen, wie das RHS-Kernel dennoch auf die Struktur des zu lösenden Anfangswertproblems eingehen kann:
  - **Unterschiedliche Mappings zwischen Workitems und Vektorkomponenten:** Hier erlaubt unsere Basisimplementierung ein 1:1, 2:1 oder 4:1 Mapping zwischen Vektorkomponenten und Workitems. Dementsprechend benötigt unsere Basisimplementierung für jedes Mapping eine spezialisierte Implementierung der RHS-Funktion, die bei einem Aufruf 1, 2 oder 4 Komponenten berechnet. Je nach Anfangswertproblem und Genauigkeit kann ein 1:1, 2:1 oder 4:1 Mapping zwischen Vektorkomponenten und Workitems auf Grund von besserem Coalescing, effizienterer Vektorisierung, der Vermeidung von redundanten Berechnungen und mehr Register-Blocking performanter sein.
  - **Unterschiedliche Dimensionen für das Kernel:** Je nach der räumlichen Struktur des zu lösenden Anfangswertproblems kann ein 1D-, 2D- oder 3D-Kernel mehr Daten über die Caches der GPU wiederverwenden.

  Beispielhaft realisierten wir für das BRUSS2D-Testproblem und das NBODY-Testproblem, die wir in dieser Arbeit experimentell untersuchen werden, folgende Kernel für die Basisimplementierung:

  - **BRUSS2D-Kernel:** Für unser BRUSS2D-Testproblem (siehe Abschnitt 5.2) verwendet unsere Basisimplementierung wegen dessen zweidimensionaler Struktur immer ein 2D-GPU-Kernel, um so die Lokalität der Speicherzugriffe von BRUSS2D zu erhöhen. Zudem verwendet das BRUSS2D-Kernel ein 4:1 Mapping zwischen Vektorkomponenten und Workitems für die einfache Gleitkommagenauigkeit, wodurch ein Workitem mit einem globalen Index von (x, y) im Iterationsraum des 2D-Kernels die beiden Gitterzellen mit den Koordinaten $(2 \cdot x,\ y)$ und $(2 \cdot x + 1,\ y)$ auswertet, und ein 2:1 Mapping für die doppelte Gleitkommagenauigkeit, wodurch ein Workitem die Gitterzelle mit den Koordinaten $(x,\ y)$ auswertet. Im BRUSS2D-Kernel erlaubt ein solches 2:1 Mapping bei doppelter Gleitkommagenauigkeit beziehungsweise ein solches 4:1 Mapping bei einfacher Gleitkommagenauigkeit jeweils, verglichen mit einem 1:1 Mapping, die Verwendung von 16-Byte-Gather- und 16-Byte-Scatter-Instruktionen mit nahezu perfektem Coalescing und eine effizientere Vektorisierung. Zudem vermeidet ein 2:1 oder ein 4:1 Mapping, dass das RHS-Kernel diejenigen Berechnungen, die sich die beiden Komponenten einer Gitterzelle teilen, redundanterweise mehrmals durchführen muss.

– **NBODY-Kernel:** Für unser NBODY-Testproblem (siehe Abschnitt 5.3) verwendet unsere Basisimplementierung ein 1D-Kernel und ein 4:1 Mapping zwischen Vektorkomponenten und Workitems. Wegen der im Abschnitt 5.3 beschriebenen Speicheranordnung des NBODY-Testproblems berechnet bei dem verwendeten 4:1 Mapping die erste Hälfte der Workitems die Beschleunigungen der Partikel und die zweite Hälfte der Workitems die Positionsänderungen der Partikel. Theoretisch könnte man ein solches NBODY-Kernel deutlich weiter optimieren, zum Beispiel, indem man das dritte Newtonsche Gesetz ausnutzt, um die Anzahl an Berechnungen des Kernels zu halbieren. Weitere Optimierungen des NBODY-Kernels sind jedoch kein Ziel dieser Arbeit, weshalb wir sie hier nicht durchführen werden.

- **(LC | MAP)-Kernel:** Für (LC | MAP)-Operationen verwendet unsere Basisimplementierung immer ein 1D-Kernel. Falls für das Lösen des Anfangswertproblems die einfache Gleitkommagenauigkeit verwendet werden soll, verwendet ein (LC | MAP)-Kernel ein 4:1 Mapping zwischen Vektorkomponenten und Workitems, wodurch für das (LC | MAP)-Kernel ein Workitem mit einem globalen Index von (x) im Iterationsraum des Kernels die vier aufeinanderfolgenden Vektorkomponenten der entsprechenden (LC | MAP)-Operation mit den Indexen $(4 \cdot x)$, $(4 \cdot x + 1)$, $(4 \cdot x + 2)$ und $(4 \cdot x + 3)$ berechnet. Falls für das Lösen des Anfangswertproblems die doppelte Gleitkommagenauigkeit verwendet werden soll, dann verwenden ein (LC | MAP)-Kernel ein 2:1 Mapping, wodurch ein Workitem mit einem globalen Index von (x) die zwei aufeinanderfolgenden Vektorkomponenten mit den Indexen $(2 \cdot x)$ und $(2 \cdot x + 1)$ berechnet. Für beide Gleitkommagenauigkeiten ermöglicht dieses Mapping ein perfektes Coalescing der Speicherzugriffe und die Verwendung von effizienten 16-Byte-Gather- und 16-Byte-Scatter-Instruktionen.
- **RED-Kernel:** Ein RED-Kernel verwendet analog zu einem (LC | MAP)-Kernel ebenfalls ein 4:1 Mapping zwischen Vektorkomponenten und Workitems für die einfache Gleitkommagenauigkeit und ein 2:1 Mapping für doppelte Gleitkommagenauigkeit. Bei diesem Kernel reduziert ein Workitem mit einem globalen Index von (x) je nach Gleitkommagenauigkeit die vier aufeinanderfolgenden Vektorkomponenten $(4 \cdot x)$, $(4 \cdot x + 1)$, $(4 \cdot x + 2)$ und $(4 \cdot x + 3)$ beziehungsweise die zwei aufeinanderfolgenden Vektorkomponenten $(2 \cdot x)$ und $(2 \cdot x + 1)$ zu einem einzigen Wert. Dann führt das Kernel eine Workgroup-weite Reduktion durch. Abschließend führt jede Workgroup mit dem Ergebnis der Workgroup-weiten Reduktion nur eine einzige atomare Operation auf eine atomare Variable im Heap-Speicher der GPU aus. Mit Hilfe der zuvor durchgeführten Workgroup-weiten Reduktion verringert das RED-Kernel den Druck auf diese atomare Variable. Letztlich gilt anzumerken, dass man dieses RED-Kernel noch deutlich weiter optimieren könnte, indem man zum Beispiel jedem Workitem mehrere Blöcke von Komponenten zuweist. Da bei ODE-Verfahren das RED-Kernel nur einen kleinen Bruchteil der Gesamtlaufzeit ausmacht, werden wir in dieser Arbeit diese weiteren Optimierungen nicht mehr durchführen.

Wir realisierten die Kernel der Basisimplementierung in CUDA-C beziehungsweise OpenCL-C. In diesen Kernels versehen wir die Zeiger auf sämtliche Kernelargumente mit dem Schlüsselwort `restrict` (siehe zum Beispiel [T9] oder [T25]). Dadurch muss der verwendete GPU-Compiler kein Aliasing zwischen den Kernelargumenten annehmen, wodurch er die Speicherzugriffe auf diese Kernelargumente stärker optimieren kann.

1D-RHS-Kernel

```
in: double2* y;
out: double2* F;

code:
parallel for each workgroup wg
   parallel for each workitem wi in wg
      int i = global_index_x(wi);
      F[i] = RHS(y, i);
```

2D-RHS-Kernel

```
in: double2* y;
out: double2* F;

code:
parallel for each workgroup wg
   parallel for each workitem wi in wg
      int i_x  =  global_index_x(wi);
      int i_y  =  global_index_y(wi);
      F[i_y * xmax + i_x] = RHS(y, i_x, i_y);
```

LC-Kernel

```
in: double2* y_k, F_1, F_2, ...;
out: double2* y_kn;

code:
parallel for each workgroup wg
   parallel for each workitem wi in wg
      int i = global_index_x(wi);
      y_kn[i] =  y_k[i] + a_1 * F_1[i] + a_2 * F_2[i] + ...;
```

RED-Kernel

```
in: double2* E;
inout: double* epsilon_global;

code:
parallel for each workgroup wg
   parallel for each workitem wi in wg
      int i = global_index_x(wi);
      double eps_wg = workgroup_reduce(E[i]);
      if(local_index_x(wi) == 0)
         atomic_max(epsilon_global, eps_wg);
```

Abbildung 8.1: Pseudocode der GPU-Kernel der Basisimplementierung.

8.3.2 Experimente

8.3.2.1 Aufbau und Durchführung der Experimente

Als Nächstes werden wir die Kernel der Basisimplementierung für GPUs experimentell per Laufzeitmessungen und Profiling untersuchen. Die Basisimplementierungen von konkreten ODE-Verfahren werden wir dagegen erst im 9. Kapitel der Kernelfusion näher untersuchen, um sie dort mit den per Kernelfusion generierten Implementierungsvarianten zu vergleichen.

Als Test-GPU wählten wir eine Geforce Volta Titan (*Volta*), deren technische Daten von Tabelle 13.2 und Tabelle 13.5 gezeigt werden. Als GPU-API verwendeten wir CUDA. Zusätzlich führten wir sämtliche Messungen für die Kernel der Basisimplementierung mit der doppelten Gleitkommagenauigkeit durch. Für die zu untersuchenden LC-Kernels wählten wir die Linearkombinationen von 2 Vektoren (*LC-2*) bis 7 Vektoren (*LC-7*) mit einer Problemgröße von 17 301 504 Komponenten. Die selbige Problemgröße wählten wir auch für unser RED-Kernel. Zudem untersuchten wir, als Beispiel für ein dünnbesetztes Anfangswertproblem, das RHS-Kernel für unser Testproblem BRUSS2D mit einer Problemgröße von $4\,224 \times 2\,048$ Gitterzellen, welche in ebenfalls 17 301 504 Komponenten resultieren. Als Beispiel für ein dichtbesetztes Anfangswertproblem mit einer hohen arithmetischen Intensität evaluierten wir das RHS-Kernel für unser NBODY-Testproblem mit einer Problemgröße von 345 856 Partikeln, welche nur 2 075 136 Komponenten entsprechen. Wir wählten diese kleinere Problemgröße für das NBODY-Testproblem, da das Profiling von dessen Kernel in CUDA ansonsten mehrere Tage gedauert hätte. Für all diese Kernels maßen wir neben der Laufzeit die *Auslastung der DRAM-Bandbreite* (U_{DRAM}), das *DRAM-Volumen* (V_{DRAM}), die *Cache-Effizienz* (E_{cache}), die *Auslastung der DP-Rechenwerke* (U_{DP}), die *Auslastung der IPC* (U_{IPC}) und die *Anzahl an herausgegebenen Instruktionen* (INS) per Profiling. Dabei beschreiben das DRAM-Volumen, wie viele Daten für ein Kernel zwischen GPU und DRAM transferiert werden, und die Cache-Effizienz, wie effizient die Caches der GPU das DRAM-Volumen unter der Annahme, dass keine Datenwiederverwendung zwischen unterschiedlichen Kernels auftritt, reduzieren können. Eine ausgiebige Beschreibung dieser Metriken wird im Abschnitt 13.4 gegeben. Die Ergebnisse der so durchgeführten Messungen werden von der Tabelle 8.1 gezeigt.

8.3.2.2 Diskussion der Messergebnisse

Für das NBODY-Kernel zeigen die Messergebnisse, dass auf Volta die Auslastung der DP-Rechenwerke mit 81.7 % sehr hoch und die Auslastung der DRAM-Bandbreite mit 0.3 % sehr niedrig ist. Dadurch ist auf Volta das NBODY-Kernel stark durch die von ihm durchgeführten Berechnungen gebunden. Ebenso dominiert das NBODY-Kernel wegen seiner quadratischen Komplexität die Laufzeit der LC-Kernel und RED-Kernel (1 049.12 ms gegenüber 0.21 ms bis 1.85 ms), und dies, obwohl wir für das NBODY-Kernel eine deutlich kleinere Problemgröße wählten. Aus diesem Grund ist auf Volta für das NBODY-Testproblem auch die gesamte Basisimplementierung eines jeden ODE-Verfahrens stark durch die von ihm durchgeführten Berechnungen gebunden. Analoges gilt auch für jedes andere dichtbesetzte Anfangswertproblem mit einer hohen arithmetischen Intensität. Folglich kann man für solche dichtbesetzten Anfangswertprobleme mit einer hohen arithmetischen Intensität für alle ODE-Verfahren unter Verwendung der Basisimplementierung bereits ohne zusätzliche Lokaliätsoptimierungen eine gute Performance erzielen. Ebenso zeigen die Messergebnisse, dass auf Volta die Cache-Effizienz mit circa 0.4 % für das NBODY-Problem sehr niedrig ist. Dies ist darauf zurückzuführen, dass für das NBODY-Problem bei der gewählten Systemgröße ein Systemzustand 16.6 MB groß ist,

Kernel	Laufzeit in [ms]	U_{DRAM} in [%]	V_{DRAM} in [GB]	E_{cache} in [%]	U_{DP} in [%]	U_{IPC} in [%]	INS in $[10^6]$
BRUSS2D	0.48	92.1	0.28	99.4	9.3	8.5	15
NBODY	1 049.12	0.3	2.42	0.4	81.7	55.2	192 520
LC-2	0.68	94.2	0.42	99.6	1.5	1.8	5
LC-3	0.89	95.8	0.55	99.3	1.5	1.7	6
LC-4	1.14	94.5	0.69	100.1	1.8	1.6	7
LC-5	1.37	93.4	0.83	99.5	1.9	1.5	8
LC-6	1.61	92.2	0.97	99.6	1.9	1.4	9
LC-7	1.85	91.1	1.11	99.8	2.0	1.1	10
RED	0.21	98.6	0.14	99.5	8.0	16.1	13

Tabelle 8.1: Laufzeitmessung und Profiling der GPU-Kernel der Basisimplementierung. Diese Tabelle zeigt die Laufzeit, die Auslastung der DRAM-Bandbreite (U_{DRAM}), das DRAM-Volumen (V_{DRAM}), die Cache-Effizienz (E_{cache}), die Auslastung der DP-Rechenwerke (U_{DP}), die Auslastung der IPC (U_{IPC}) und die Anzahl an herausgegebenen Instruktionen (INS) des jeweiligen Kernels. Eine detaillierte Beschreibungen dieser Metriken wird in Abschnitt 13.4 gegeben. Diese Messungen wurden auf Volta durchgeführt.

während der L2-Cache von Volta nur 6 MiB groß ist. Dadurch kann Volta nicht den gesamten Systemzustand des NBODY-Problems in ihrem L2-Cache abspeichern. Da für die Berechnung der Partikelinteraktionen jeder Thread einmal über sämtliche Partikel iterieren muss, muss Volta den Zustand jedes Partikels mehrmals aus dem DRAM laden, wodurch sich wiederum die niedrige Cache-Effizienz ergibt.

Dahingegen liegt auf Volta für alle LC-Kernel, das RED-Kernel, und das BRUSS2D-Kernel die Auslastung der DRAM-Bandbreite zwischen 91.1 % und 98.6 % sowie die Auslastung der DP-Einheiten zwischen 2.0 % und 9.3 %. Folglich sind im Falle eines dünnbesetzten Anfangswertproblems mit einer niedrigen arithmetischen Intensität, wie bei unserem BRUSS2D-Testproblem, auf Volta alle Kernel sehr stark durch die DRAM-Bandbreite gebunden. Da bei den für diese Arbeit relevanten Problemgrößen ein Vektor deutlich größer als die Caches der GPU ist (für diese Untersuchung wurde beispielhaft eine Vektorgröße von 132 MiByte gewählt, während der L2-Cache von Volta nur 6 MiByte groß ist), gibt es auch kaum Datenwiederverwendung, wenn zwei nacheinander ausgeführte Kernels auf denselben permanenten Vektor zugreifen. Dadurch bleibt in diesem Fall auch das als zweites ausgeführte Kernel durch die DRAM-Bandbreite gebunden. Aus diesem Grund ist auf GPUs das Lösen eines dünnbesetzten Anfangswertproblems mit einer niedrigen arithmetischen Intensität durch die Basisimplementierung für alle ODE-Verfahren ebenfalls durch die DRAM-Bandbreite gebunden. Somit ist es für dünnbesetzte Anfangswertprobleme mit einer niedrigen arithmetischen Intensität vielversprechend, die Basisimplementierung durch Lokalitätsoptimierungen zu verbessern. Allerdings zeigt auf Volta die Cache-Effizienz von nahezu 100 % auch, dass das Caching innerhalb jeder dieser Kernels das DRAM-Volumen beinahe auf das Datenvolumen reduziert.

8.4 Basisimplementierung auf CPUs

8.4.1 Implementierung

In unserer Basisimplementierung für CPUs übersetzen wir jedes abstrakte Kernel in ein paralleles Schleifennest, welches wir im Folgenden als *CPU-Kernel* bezeichnen. Ein solches CPU-Kernel besteht aus einer parallelen äußeren Schleife, die über die Blöcke des Kernels iteriert, und aus einer inneren Schleifenstruktur, die von der Grundoperation des Kernels abhängig ist. Da wir diese CPU-Kernel der Basisimplementierung nicht allgemein für eine beliebige CPU-Architektur optimieren können, überlegen wir uns im Folgenden nur, wie wir diese CPU-Kernel für moderne x64-Server-CPUs, wie sie zum Beispiel im Abschnitt 3.14 und von den Werken [T22], [T23] und [W15] beschrieben werden, optimieren können. Dabei berücksichtigen wir insbesondere die Vektorisierung, und wie sich mit Non-Temporal-Store-Instruktionen der Fetch-On-Write-Overhead des Last-Level-Caches vermeiden lässt. Allerdings lassen sich analoge Überlegungen auch verwenden, um die CPU-Kernels der Basisimplementierung für andere CPU-Architekturen zu optimieren. Für unsere CPU-Kernels der Basisimplementierung nehmen wir wieder große Systemgrößen für die zu lösenden Anfangswertprobleme an, wodurch es keine oder nur sehr wenig Datenwiederverwendung zwischen zwei aufeinanderfolgenden Kernels gibt. Dadurch müssen die Kernels, um den Fetch-on-Write-Overhead des Last-Level-Caches der CPU zu vermeiden, Non-Temporal-Store-Instruktionen für das Zurückschreiben von Vektorkomponenten in die permanenten Vektoren verwenden. Die innere Schleifenstruktur der CPU-Kernels der Basisimplementierung realisieren wir wie folgt (siehe Abbildung 8.2 für einen Pseudocode der CPU-Kernels):

- **RHS-Kernel:** Für ein RHS-Kernel unterstützen wir zwei unterschiedliche Implementierungen der RHS-Funktion, die jeweils ein CPU-Kernel mit einer anderen inneren Schleifenstruktur benötigen:
  - **Eval-Comp-Funktion:** Ein Aufruf der *Eval-Comp-Funktion* wertet die RHS-Funktion für eine einzige gegebene Komponente aus. Folglich benötigt ein CPU-Kernel für die Eval-Comp-Funktion eine innere Schleife, die über die Komponenten des entsprechenden Blocks mit einem Stride von Eins iteriert. Dabei kann der Compiler leicht den Körper der Eval-Comp-Funktion per Inline-Ersetzung in diese innere Schleife einbauen und zudem diese innere Schleife abrollen. Jedoch kann die innere Struktur der Eval-Comp-Funktion verhindern, dass der Compiler nach der Inline-Ersetzung über aufeinanderfolgende Iterationen der inneren Schleife des RHS-Kernels vektorisieren kann. Zuletzt führt die komponentenweise Auswertung der RHS-Funktion bei vielen Problemen redundante Berechnungen durch, wie zum Beispiel die Randbehandlung, welche der Compiler nur unter Umständen erkennen und entfernen kann. Ein RHS-Kernel mit der Eval-Comp-Funktion verwendet für das Zurückschreiben seines Ergebnisses in seinen permanenten Vektor Non-Temporal-Store-Instruktionen.
  - **Eval-Range-Funktion:** Ein Aufruf der *Eval-Range-Funktion* wertet die RHS-Funktion für eine Anzahl von gegebenen aufeinanderfolgenden Komponenten aus. Dabei iteriert die Eval-Range-Funktion mit einer inneren Schleifenstruktur über die auszuwertenden Komponenten. Diese innere Schleifenstruktur kann jedoch auf das entsprechende Anfangswertproblem auf eine Weise zugeschnitten werden, so dass der Compiler sie leicht vektorisieren und abrollen kann. Alternativ kann die innere

Schleifenstruktur dem Compiler ihre Vektorisierung über Intrinsics bereits vorgeben. Zusätzlich kann die innere Schleifenstruktur der Eval-Range-Funktion beinahe alle redundanten Berechnungen der Eval-Comp-Funktion vermeiden. Folglich muss bei dieser Implementierung das RHS-Kernel für jeden seiner Blöcke nur einmal die Eval-Range-Funktion aufrufen. Da die Eval-Range-Funktion ihr Ergebnis bereits direkt in einem permanenten Vektor abspeichert, muss eine Implementierung einer Eval-Range-Funktion eigenverantwortlich zur Vermeidung des Fetch-on-Write-Overheads auf Non-Temporal-Store-Instruktionen zurückgreifen.

- **(LC | MAP)-Kernel:** Bei einem (LC | MAP)-Kernel iteriert die innere Schleife über die Komponenten des entsprechenden Blocks mit einem Stride von Eins und berechnet die Linearkombination für den entsprechenden Vektorkomponenten oder ruft die MAP-Funktion für den entsprechenden Vektorkomponenten auf. Dadurch kann der Compiler die innere Schleife leicht per Loop-Unrolling oder Vektorisierung (siehe zum Beispiel [T26]) optimieren. Zusätzlich verwendet auch ein (LC | MAP)-Kernel für das Zurückschreiben seines Ergebnisses in seinen permanenten Vektor Non-Temporal-Store-Instruktionen.

- **RED-Kernel:** In einem RED-Kernel führt ein jeder Thread zunächst eine partielle lineare Reduktion der Komponenten seiner Blöcke aus, indem er über die Komponenten seiner Blöcke mit einem Stride von Eins iteriert. Nachdem die Threads über alle Blöcke iteriert haben, werden die Zwischenergebnisse aller Threads noch einmal zum Endergebnis der Reduktion zusammengeführt. Während die Gleitkomma-Arithmetik es einem Compiler prinzipiell erlaubt, eine lineare Minimums- oder Maximums-Reduktion zu vektorisieren, so erlaubt sie es dem Compiler wegen potentiellen Rundungsfehlern jedoch nicht, eine lineare Additionsreduktion zu vektorisieren. Während wir das RED-Kernel unserer Basisimplementierung für CPUs noch weiter optimieren könnten, so verzichten wir in dieser Arbeit wieder darauf, da RED-Operationen typischerweise nur einen sehr kleinen Anteil zu der Gesamtlaufzeit für die Lösung eines Anfangswertproblems beitragen.

Wir implementierten die Kernel der Basisimplementierung in der Programmiersprache C. In dem Quelltext der Kernels realisierten wir die Non-Temporal-Store-Instruktionen je nach Zielsystem über AVX2-Intrinsics oder AVX-512-Intrinsics, wie sie in [T23] beschrieben werden. Zusätzlich versahen wir wieder die Zeiger auf sämtliche Kernelargumente mit dem Schlüsselwort `restrict`, damit der verwendete C-Compiler potentielles Aliasing zwischen den Kernelargumenten ausschließen und dadurch wiederum die Zugriffe auf diese stärker optimieren kann (siehe zum Beispiel [W16] für eine Behandlung dieser Thematik).

RHS-Eval-Comp-Kernel

```
in: double* y;
out: double* F;

code:
parallel for each block b
   for each component i in b
      F[i] <- RHS_Eval_Comp(y, i);
```

RHS-Eval-Range-Kernel

```
in: double* y;
out: double* F;

code:
parallel for each block b
   RHS_Eval_Range(F, y,  b, lblock);
```

LC-Kernel

```
in: double* y_k, F_1, F_2, ...;
out: double* y_kn;

code:
parallel for each block b
   for each component i in b
      y_kn[i] <-  y_k[i] + a_1 * F_1[i] + a_2 * F_2[i] + ...;
```

RED-Kernel

```
in: double2* E;
inout: double* epsilon_global;

code:
double epsilon_local = DOUBLE_MIN;
parallel for each block b
   for each component i in b
      epsilon_local = max(E[i], epsilon_local);
atomic_max(epsilon_global, epsilon_local);
```

Abbildung 8.2: Pseudocode der CPU-Kernel der Basisimplementierung. Hierbei symbolisiert der Operator <− Non-Temporal-Store-Instruktionen, durch welche Vektorkomponenten direkt in den DRAM geschrieben werden.

8.4.2 Experimente

8.4.2.1 Aufbau und Durchführung der Experimente

Als Nächstes werden wir wieder die Kernel der Basisimplementierung für CPUs und mit der doppelten Gleitkommagenauigkeit per Laufzeitmessungen und Profiling untersuchen. Dagegen werden wir die Basisimplementierungen von konkreten ODE-Verfahren erst im 9. Kapitel der Kernelfusion näher auf CPUs untersuchen, um sie dort mit den per Kernelfusion generierten Implementierungsvarianten zu vergleichen.

Für die experimentelle Untersuchung der Kernel der Basisimplementierung wählten wir als Test-CPU einen Intel Core i9-9980XE CPU (*Skylake*), deren technische Daten im Abschnitt 13.6 gezeigt werden. Wir kombinierten diese Skylake-CPU für unsere Messungen mit 4 $\times$ 16GiB-DDR4-PC17000-Modulen, wodurch sich eine Burst-DRAM-Bandbreite von 68 GB/s ergibt. Als C-Compiler für die Kernel der Basisimplementierung verwendeten wir den *ICC-Compiler* von Intel [W17], da dieser sehr gut optimierten Maschinencode für Intel-Architekturen erzeugt. Als Compiler-Optionen verwendeten wir die Optimierungsstufe `-O3` sowie die beiden Hinweise, spezifisch Maschinencode für die Skylake-Architektur zu erzeugen (`-march=skylake-avx512`) und zusätzlich eine 512-Bit-Vektorisierung zu bevorzugen (`-qopt-zmm-usage=high`). Zudem stellten wir durch Assembly-Analyse sicher, dass der Intel-Compiler sämtliche Kernel erfolgreich vektorisierte. Damit die Experimente für die Basisimplementierung auf GPUs und CPUs miteinander vergleichbar sind, führten wir die CPU-Experimente auf Skylake wiederum mit der doppelten Gleitkommagenauigkeit und mit derselben Problemgröße wie die GPU-Experimente durch. So evaluierten wir auf Skylake wiederum die LC-Kernel der Linearkombination von 2 Vektoren (LC-2) bis 7 Vektoren (LC-7) mit einer Problemgröße von 17 301 504 Komponenten und das RED-Kernel mit der selbigen Problemgröße. Zudem wählten wir auf Skylake wieder als Beispiel für das RHS-Kernel eines dünnbesetzten Anfangswertproblem mit einer niedrigen arithmetischen Intensität das RHS-Kernel unseres Testproblems BRUSS2D mit einer Problemgröße von 4 224 $\times$ 2 048 Gitterzellen, welche ebenfalls in 17 301 504 Komponenten resultieren. Ebenso evaluierten wir auf Skylake wieder als Beispiel für ein dichtbesetztes Anfangswertproblem mit einer hohen arithmetischen Intensität das RHS-Kernel für unser Testproblem NBODY mit einer Problemgröße von 345 856 Partikeln, welche nur 2 075 136 Komponenten entsprechen. Für all diese Kernels maßen wir neben der Laufzeit die *Auslastung der DRAM-Bandbreite* (U_{DRAM}), das *DRAM-Volumen* (V_{DRAM}), die *Cache-Effizienz* (E_{cache}), die *Ausnutzung der DP-Peak-Performance* ($U_{\text{DP-PEAK}}$), die *Auslastung der IPC* (U_{IPC}) und die *Anzahl an herausgegebenen Instruktionen* (INS) per Profiling. Diese Metriken werden im Abschnitt 13.4 noch einmal ausführlich beschrieben. Die Ergebnisse der so durchgeführten Messungen werden von der Tabelle 8.2 gezeigt.

8.4.2.2 Diskussion der Messergebnisse

Die Messergebnisse auf der Skylake-CPU zeigen ein sehr ähnliches Bild wie die Messergebnisse auf der Volta-GPU:

So beträgt die Laufzeit des NBODY-Kernels 10 573.1 ms, während die Laufzeit der anderen Kernels nur zwischen 2.5 und 22.0 ms beträgt. Dadurch dominiert das NBODY-Kernel auf Skylake genauso wie auf Volta die Laufzeit aller anderen Kernel. Zusätzlich lastet das NBODY-Kernel die Burst-DRAM-Bandbreite von Skylake nur zu 2 % aus, das heißt, es ist genauso wie auf Volta nicht durch die DRAM-Bandbreite gebunden. Allerdings nutzt das NBODY-Kernel

Kernel	Laufzeit in [ms]	U_{DRAM} in [%]	V_{DRAM} in [GB]	E_{cache} in [%]	$U_{\text{DP-PEAK}}$ in [%]	U_{IPC} in [%]	INS in $[10^6]$
BRUSS2D	6.7	65.8	0.30	92.4	1.9	4.6	91
NBODY	10 573.1	1.9	1.39	0.4	12.9	11.4	468 601
LC-2	7.3	80.2	0.40	104.7	0.4	0.5	14
LC-3	12.2	69.4	0.57	96.0	0.4	0.5	17
LC-4	14.2	74.7	0.72	96.1	0.5	0.5	21
LC-5	16.5	76.8	0.86	96.4	0.6	0.5	23
LC-6	18.0	81.1	0.99	97.6	0.7	0.4	26
LC-7	22.0	77.8	1.15	96.3	0.7	0.4	28
RED	2.5	76.0	0.13	105.4	0.1	1.0	10

Tabelle 8.2: Laufzeitmessung und Profiling der CPU-Kernel der Basisimplementierung. Die Tabelle zeigt die Laufzeit, die Auslastung der DRAM-Bandbreite (U_{DRAM}), das DRAM-Volumen (V_{DRAM}), die Cache-Effizienz (E_{cache}), die Ausnutzung der DP-Peak-Performance ($U_{\text{DP-PEAK}}$), die Auslastung der IPC (U_{IPC}) und die Anzahl an herausgegebenen Instruktionen (INS) des jeweiligen Kernels. Eine detaillierte Beschreibungen dieser Metriken wird in Abschnitt 13.4 gegeben. Diese Messungen wurden auf Skylake durchgeführt.

auf Skylake dessen Peak-DP-Performance nur zu 12.9 % aus, während das NBODY-Kernel auf Volta die DP-Recheneinheiten zu 81.7 % auslastet. Dies führt dazu, dass, obwohl Volta eine 3.8-mal so hohe Peak-DP-Performance wie Skylake besitzt (6.10 TFLOPS auf Volta gegenüber 1.61 TFLOPS auf Skylake), das NBODY-Kernel auf Volta circa 10.1-mal so schnell ist wie auf Skylake (1 049 ms auf Volta gegenüber 10 573 ms auf Skylake). Zusätzliches Profiling zeigt, dass unsere Implementierung des NBODY-Kernels auf Skylake durch die Cache-Bandbreite gebunden ist. Allerdings realisierten wir für unsere CPU-Implementierung die Eval-Range-Funktion des NBODY-Kernels auch naiv über zwei ineinander verschachtelte For-Schleifen. Es ist jedoch noch möglich, diese naive Implementierung des NBODY-Kernels noch weiter zu optimieren, so dass sie die Rechenleistung einer modernen x64-Server-CPU deutlich besser ausnutzen kann und auf Grund ihrer hohen arithmetischen Intensität auch durch die Rechenleistung der CPU gebunden wird. Solche Optimierungen für eine CPU-Implementierung des NBODY-Kernels werden zum Beispiel von der Arbeit [K34] vorgestellt und untersucht.

Das BRUSS2D-Kernel, die LC-Kernels und das RED-Kernel nutzen die DP-Rechenleistung von Skylake nur zwischen 0.1 % und 1.9 % aus. Zudem lasten diese Kernels die Burst-DRAM-Bandbreite von Skylake zwischen 65.8 % und 81.1 % aus. Dadurch sind das BRUSS2D-Kernel, die LC-Kernels und das RED-Kernel auf Skylake genauso wie auf Volta durch die DRAM-Bandbreite gebunden. Des Weiteren besitzen diese Kernels auf Skylake eine Cache-Effizienz zwischen 92 % und 105 %. Daraus können wir folgern, dass für diese Kernels auf Skylake genauso wie auf Volta die Caches das DRAM-Volumen beinahe auf das Datenvolumen reduzieren können. Zusätzlich zeigt diese hohe Cache-Effizienz, dass wir über die Non-Temporal-

Store-Instruktionen erfolgreich den Fetch-On-Write-Overhead des Last-Level-Caches verhindern können. Da Skylake eine Burst-DRAM-Bandbreite von 68 GB/s besitzt, während Volta eine Burst-DRAM-Bandbreite von 653 GB/s besitzt, würden wir erwarten, dass diese durch die DRAM-Bandbreite gebundenen Kernels auf Skylake ungefähr eine 9.6-mal so lange Laufzeit benötigen wie auf Volta. Tatsächlich zeigen die Messungen für die DRAM-gebundenen Kernels auf Skylake eine in etwa 11- bis 15-mal so lange Laufzeit wie auf Volta auf. Die Abweichung der Messergebnisse von den Erwartungen ist darauf zurückzuführen, dass die vom Skylake verwendete DDR4-Technologie verglichen mit der von Volta verwendeten HBM-Technologie nur einen kleineren Anteil ihrer Burst-DRAM-Bandbreite effektiv ausnutzen kann. So beträgt auf Skylake die Auslastung der Burst-DRAM-Bandbreite bei den (BRUSS2D | LC | RED)-Kernels zwischen 70 % und 80 %, während die Auslastung der Burst-DRAM-Bandbreite auf Volta bei den (BRUSS2D | LC | RED)-Kernels zwischen 90 % und 100 % beträgt. Eine kurze technische Begründung hierfür werden wir in den Abschnitten 13.4 und 13.5 gegeben.

Wenn wir die Anzahl der für ein Kernel ausgeführten Instruktionen zwischen Skylake und Volta miteinander vergleichen, so würden wir in etwa ein Verhältnis von 0.25:1 erwarten, da Skylake mit AVX-512 bei doppelter Gleitkommagenauigkeit eine logische Vektorbreite von 8 Komponenten besitzt, während Volta bei doppelter Gleitkommagenauigkeit eine logische Vektorbreite von 32 Komponenten besitzt. Bei den LC-Kernels beträgt das Verhältnis zwischen den von Skylake und Volta ausgeführten Instruktionen circa 0.34:1, für das BRUSS2D-Kernel 0.16:1, für das RED-Kernel 0.77:1, und bei dem NBODY-Kernel 0.41:1. Somit entspricht das Verhältnis der Anzahl an ausgeführten Instruktionen für sämtliche Kernels auch in etwa den Erwartungen.

Auf Grund dieser Beobachtungen können wir für die Basisimplementierung auf CPUs dieselben Schlussfolgerungen wie für die Basisimplementierung auf GPUs ziehen. Da für dichtbesetzte Anfangswertprobleme das RHS-Kernel beziehungsweise die RHS-Operationen die Laufzeit auch auf einer CPU dominieren, ist es, um für solche dichtbesetzten Anfangswertprobleme durch die Basisimplementierung eine gute Performance zu erzielen, nur nötig, die Performance des RHS-Kernels zu optimieren. Dahingegen besitzt für dichtbesetzte Anfangswertprobleme die Datenwiederverwendung zwischen unterschiedlichen Vektorgrundoperationen nur einen vernachlässigbaren Einfluss auf die Performance. Auf CPUs sind im Gegensatz zu dem NBODY-Kernel die LC-Kernel, das RED-Kernel, und das BRUSS2D-Kernel durch die DRAM-Bandbreite gebunden. Da bei den für diese Arbeit relevanten Problemgrößen ein permanenter Vektor deutlich größer als die Caches der CPU ist (für diese Untersuchung wählten wir beispielhaft eine Vektorgröße von 132 MiByte, während zum Beispiel der L3-Cache von Skylake nur 24.75 MiByte groß ist), gibt es auch auf CPUs kaum Datenwiederverwendung, wenn zwei nacheinander ausgeführte Kernels auf denselben permanenten Vektor zugreifen. Dadurch bleibt in diesem Fall auch das als zweites ausgeführte Kernel auf der CPU durch die DRAM-Bandbreite gebunden. Aus diesem Grund ist das Lösen eines dünnbesetzten Anfangswertproblems mit einer niedrigen arithmetischen Intensität durch die Basisimplementierung für alle ODE-Verfahren auf CPUs ebenfalls durch die DRAM-Bandbreite gebunden. Somit ist es für dünnbesetzte Anfangswertprobleme auch auf CPUs vielversprechend, die Basisimplementierung durch Lokalitätsoptimierungen zu verbessern.

8.5 Fazit

In diesem Kapitel zeigten wir, dass wir eine Basisimplementierung von ODE-Verfahren leicht aus deren Datenflussgraphrepräsentation erzeugen können. Diese Basisimplementierung startet für jede Vektorgrundoperation ein paralleles abstraktes Kernel auf einer gegebenen Zielplattform. Auf GPUs lässt sich jedes abstrakte Kernel effizient in ein GPU-Kernel übersetzen, während auf CPUs sich jedes abstrakte Kernel effizient in ein paralleles Schleifennest übersetzen lässt. Auf diese Weise kann die Basisimplementierung innerhalb ihrer Kernel die Systemparallelität der Vektorgrundoperationen des ODE-Verfahrens ausnutzen, um so die Berechnungen des ODE-Verfahrens parallel zueinander auf sämtlichen Kernen des Zielrechners auszuführen. Unsere anschließenden Untersuchungen zeigten, dass diese Basisimplementierung für dichtbesetzte Anfangswertprobleme mit einer hohen arithmetischen Intensität sowohl auf GPUs als auch auf CPUs bereits deren Rechenleistung gut ausnutzen und somit eine gute Performance erzielen kann. Allerdings zeigten unseren Untersuchungen auch, dass für dünnbesetzte Anfangswertprobleme mit einer niedrigen arithmetischen Intensität die Basisimplementierung für sämtliche ODE-Verfahren sowohl auf GPUs als auch auf CPUs durch die DRAM-Bandbreite gebunden ist.

9 Kernelfusion

9.1 Motivation

Wie die Messungen im vorherigen Kapitel zeigten, so dominieren für dichtbesetzte Anfangswertprobleme RHS-Operationen auf Grund ihrer quadratischen Rechenkomplexität die Laufzeit von jeder möglichen Implementierung eines ODE-Verfahrens. Besitzt ein dichtbesetztes Anfangswertproblem zusätzlich eine hohe arithmetische Intensität, wie es zum Beispiel bei unserem NBODY-Testproblem der Fall ist, dann kann die Basisimplementierung eines jeden ODE-Verfahrens für ein solches Anfangswertproblem bereits eine sehr gute Performance erzielen. Deshalb ist es auch für dichtbesetzte Anfangswertprobleme nicht nötig, die Lokalität der Speicherzugriffe der Basisimplementierung zu optimieren.

Im Gegensatz dazu ist die Basisimplementierung für dünnbesetzte Anfangswertprobleme mit einer niedrigen arithmetischen Intensität, wie zum Beispiel viele Stencils, nur bei kleinen Problemgrößen, bei welchen die benötigten permanenten Vektoren noch in den Caches des Prozessors Platz finden und ein Prozessor deshalb zwischen zwei aufeinanderfolgenden Vektorgrundoperationen viele Daten über seine Caches wiederverwenden kann, durch die Rechenleistung des Prozessors gebunden. Jedoch passen bereits bei mittelgroßen Problemgrößen die benötigten permanenten Vektoren der Basisimplementierung nicht mehr in die Caches des Prozessors. Dadurch existiert zwischen zwei aufeinanderfolgenden Vektorgrundoperationen beziehungsweise Kernels der Basisimplementierung nur sehr wenig oder keinerlei Datenwiederverwendung mehr. Da zudem bei dünnbesetzten Anfangswertproblemen mit einer niedrigen arithmetischen Intensität jede Art von Vektorgrundoperationen beziehungsweise jedes Kernel der Basisimplementierung nur wenige Berechnungen durchführt, ist für solche Anfangswertprobleme die Basisimplementierung bereits bei einer mittelgroßen Problemgröße durch die DRAM-Bandbreite gebunden. Aus diesem Grund ist es für dünnbesetzte Anfangswertprobleme mit einer niedrigen arithmetischen Intensität bei einer mittelgroßen oder großen Problemgröße für die Laufzeit entscheidend, möglichst viele Daten zwischen den Vektorgrundoperationen über die On-Chip-Speicher des Prozessors wiederzuverwenden. Deshalb scheint es vielversprechend zu sein, die Basisimplementierung durch Techniken zur Erhöhung der Datenwiederverwendung zu verbessern.

9.2 Theorie

9.2.1 Überblick

Um die Datenwiederverwendung der Basisimplementierung zu erhöhen, entschieden wir uns für die Technik der *Kernelfusion*. Diese Technik der Kernelfusion ähnelt der traditionellen Technik der *Fusion von Schleifen*. Jedoch abstrahieren beide Techniken das zu optimierende Problem unterschiedlich: Die Kernelfusion abstrahiert ein gegebenes Problem über einen Datenflussgraphen, der aus problemspezifischen Grundoperationen besteht, und verschmilzt mehrere dieser Grundoperationen zu größeren fusionierten Kernels, um dadurch mehr Datenwiederverwendung innerhalb eines solchen fusionierten Kernels zu erzielen. Somit handelt es sich bei diesen fusionierten Kernels um eine Partitionierung des Datenflussgraphen. Dahingegen betrachtet die Fusion von Schleifen ein gegebenes Programm, das aus mehreren Schleifennestern besteht, deren Datenwiederverwendung es durch deren Verschmelzung zu optimieren gilt.

Dementsprechend abstrahiert unser Ansatz der Kernelfusion für ODE-Verfahren ein gegebenes ODE-Verfahren zunächst durch seine Datenflussgraphräpresentation, wie sie im Kapitel 6 beschrieben wurde. Als Nächstes verschmilzt unser Ansatz der Kernelfusion die Vektorgrundoperationen in dem Datenflussgraphen zu größeren fusionierten Kernels. Jedoch erlaubt das Speicherzugriffsmuster der Vektorgrundoperationen keine beliebigen Fusionen, sondern die Vektorgrundoperation sind nur nach bestimmten Regeln fusionierbar, welche unser Ansatz für seine Korrektheit beachten muss.

Unser Ansatz der Kernelfusion erfordert zunächst, dass für alle Vektorgrundoperationen in einem fusionierten Kernel das gleiche Mapping zwischen den Vektorkomponenten und den parallelen Blöcken des Kernels existieren muss. Dadurch kann bei unserem Ansatz ein Block eines fusionierten Kernels den On-Chip-Speicher für den Datentransfer zwischen zwei fusionierten Vektorgrundoperationen verwenden. Ebenso kann ein Block eines fusioniertes Kernel den On-Chip-Speicher dafür verwenden, dass er diejenigen permanenten Vektoren, die von mehreren seiner fusionierten Vektorgrundoperationen als Argument gelesen werden, nur einmal aus dem DRAM in den On-Chip-Seicher laden muss. Zusätzlich muss bei unserem Ansatz der Kernelfusion ein Block eines fusionierten Kernels die von ihm produzierten Vektorkomponenten, die von keinem späteren Kernel mehr gelesen werden, nicht von dem On-Chip-Speicher in einen permanenten Vektor beziehungsweise in den DRAM zurückschreiben. Zwischen zwei aufeinanderfolgenden Kernels kann ein Prozessor jedoch selbst bei einer per Kernelfusion generierten Implementierungsvariante weiterhin nur bei kleinen ODE-Systemen Daten über seine Caches wiederverwenden. Somit transferiert bei unserem Ansatz der Kernelfusion ein Prozessor sämtliche Daten innerhalb eines Kernels immer über den On-Chip-Speicher, während er sämtliche Daten zwischen zwei aufeinanderfolgenden Kernels bereits bei mittelgroßen ODE-Systemen über den DRAM transferiert. Unser Ansatz verwendet zusätzlich ein Performance-Modell, welches die Laufzeit einer fusionierten Variante anhand deren Datenvolumen abschätzt. Zusätzlich wendet unser Ansatz vor der Fusion Enabling-Transformationen an, und zwar eine Cloning-Transformation, eine Splitting-Transformation und eine Move-Transformation, welche zusätzliche Fusionen von Vektorgrundoperationen ermöglichen und so die Datenwiederverwendung weiter erhöhen.

Als Nächstes erzeugt unser Ansatz der Kernelfusion für jedes fusioniertes Kernel über einen automatisierten Codegenerator jeweils ein ausführbares Programm für den Zielrechner, das auf die Architektur des Zielrechners optimiert ist. Danach kann unser Ansatz mit der Zeitschrittprozedur beginnen, welche die Zeitschritte des ODE-Verfahrens auf dem Zielrechner berechnet. Für jeden Zeitschritt führt die Zeitschrittprozedur analog wie bei der Basisimplementierung sämtliche ausführbaren Objekte in dem Datenflussgraphen, also die Konstantengenerierung, die Programme der fusionierten Kernels und die Schrittweitenkontrolle, in ihrer topologischen Reihenfolge gemäß ihren Abhängigkeiten untereinander aus. Dabei berechnet in einem solchen fusionierten Kernel wiederum jeder parallele Block die ihm zugewiesenen Komponenten der fusionierten Vektorgrundoperationen gemäß ihrer topologischen Reihenfolge.

9.2.2 Annahmen zur Kernelfusion für ein Performance-Modell

Als Nächstes werden wir ein einfaches Performance-Modell entwickeln, um die Laufzeit der fusionierten Kernels und damit die Laufzeit einer per Kernelfusion generierten Implementierungsvariante eines ODE-Verfahrens abzuschätzen. Für dieses Performance-Modell und für die weiteren zusätzlichen theoretischen Überlegungen in diesem Kapitel treffen wir folgende Annahmen:

- **Große Problemgrößen:** Die Größe des zu lösenden Anfangswertproblems sei deutlich größer als die Caches des Prozessors. Dadurch kann ein Kernel nur einen sehr kleinen Prozentsatz der Daten seines Vorgängerkernels über die Caches des Prozessors wiederverwenden.
- **Speichergebunden:** Sowohl die unfusionierten Varianten der Basisimplementierung als auch die per Kernelfusion erzeugten Varianten seien durch Speicherbandbreite beziehungsweise durch die DRAM-Bandbreite gebunden.
- **Perfektes Caching:** Innerhalb eines Kernels funktioniere das Caching perfekt, wodurch sich für die Metrik der *Cache-Effizienz* wiederum ein Wert von ungefähr $100\,\%$ ergibt (siehe Abschnitt 13.4 für eine Erklärung dieser Metrik). Dieses perfekte Caching bedeutet, dass, falls ein Kernel mehrmals auf eine Komponente eines permanenten Vektors im DRAM zugreift, dann lädt das Kernel diese Komponente beim erstmaligen Zugriff in die Caches beziehungsweise in den On-Chip-Speicher des Prozessors, welcher dann alle darauffolgenden Zugriffe auf diese Komponente über diesen On-Chip-Speicher abwickelt. Zusätzlich darf bei einem perfekten Caching der Last-Level-Cache des Prozessors beim Schreiben in einen permanenten Vektor beziehungsweise in den DRAM keinen Overhead durch eine Write-Allocate-Policy mit Fetch-On-Write verursachen. Auch müssen für ein perfektes Caching die On-Chip-Speicher groß genug sein, um zusätzlich sämtliche temporären Zwischenergebnisse des Kernels zwischenzuspeichern, so dass der Prozessor diese Zwischenergebnisse nicht in den DRAM auslagern muss.

Diese Annahmen können wir nun verwenden, um zunächst die Laufzeit eines Kernels über dessen *Datenvolumen* und darüber dann die Laufzeit eines Zeitschritts einer per Kernelfusion generierten Implementierungsvariante oder einer Basisimplementierung abzuschätzen. Dabei ist das Datenvolumen definiert als diejenige Datenmenge, die der Prozessor für die Ausführung des Kernels oder eines Zeitschritts bei einem perfekten Caching aus dem DRAM laden und in den DRAM schreiben muss (siehe Abschnitt 13.4 für eine nähere Erläuterung der Metrik des Datenvolumens). Insgesamt lässt sich unser Performance-Modell dadurch als eine Vereinfachung des Roofline-Modells interpretieren, da dieses Roofline-Modell nicht nur das Datenvolumen, sondern zusätzlich auch die arithmetische Intensität für die Performance-Abschätzung verwendet.

Unter diesen soeben genannten Annahmen ergibt sich die Laufzeit eines Kernels k aus dessen Datenvolumen geteilt durch die DRAM-Bandbreite $w_{\text{max},k}$, die das Kernel maximal auf einem Prozessor erreichen kann:

$$T_k = \frac{V_{\text{data},k}}{w_{\text{max},k}}. \tag{9.1}$$

Dabei beträgt diese maximale DRAM-Bandbreite $w_{\text{max},k}$ je nach Prozessor, DRAM-Technologie und Kernel zwischen $70\,\%$ und $95\,\%$ der Burst-DRAM-Bandbreite (siehe Abschnitt 13.4). Da ein fusioniertes Kernel k nur komplette Vektoren aus dem DRAM liest oder in den DRAM zurückschreibt, berechnet sich das Datenvolumen dieses Kernels aus der Summe der Anzahl an gelesenen Vektoren L_k und geschriebenen Vektoren S_k multipliziert mit der Größe B eines Vektors in Byte:

$$V_{\text{data},k} = (L_k + S_k) \cdot B. \tag{9.2}$$

Dabei lässt sich diese Größe B wiederum aus der Problemgröße n des zu lösenden Anfangswertproblems und der Gleitkommagenauigkeit errechnen:

$$B = n \cdot \text{sizeof(precision)}. \tag{9.3}$$

Da auf Grund der großen Problemgröße ein Prozessor nur vernachlässigbar viele Vektorkomponenten zwischen zwei aufeinanderfolgenden Kernels über seine On-Chip-Speicher wiederverwenden kann, gilt zusätzlich, dass bei einer per Kernelfusion generierten Implementierungsvariante oder bei einer Basisimplementierung die Gesamtlaufzeit eines Zeitschritts mit K Kernels ungefähr die Summe der Einzellaufzeiten all dieser K Kernels ist:

$$T_{\text{gesamt}} = \sum_{k}^{K} T_k. \tag{9.4}$$

Dadurch ist die Gesamtlaufzeit eines Zeitschritts direkt proportional zum Datenvolumen all seiner Kernels. Dieses Datenvolumen ist wiederum direkt proportional zur Summe der Anzahl an gelesenen und geschriebenen Vektoren über all diese K Kernels:

$$T_{\text{gesamt}} \propto \sum_{k}^{K} \left(L_k + S_k\right). \tag{9.5}$$

Somit können wir die Laufzeit einer per Kernelfusion erzeugten Implementierungsvariante optimieren, wenn wir über die Kernelfusion die Summe der Anzahl an gelesenen und geschriebenen Vektoren minimieren.

Für denjenigen Fall, dass auf einem Prozessor der Last-Level-Cache eine Write-Allocate-Policy mit Fetch-on-Write implementiert, können wir einen analogen Ansatz zur Abschätzung der Performance verwenden. Dieser Ansatz muss aber, um den Fetch-On-Write-Overhead zu berücksichtigen, die Anzahl an geschriebenen Vektoren doppelt gewichten. Dadurch gilt für die Laufzeit eines Zeitschritts:

$$T_{\text{gesamt mit FoW-Overhead}} \propto \sum_{k}^{K} \left(L_k + 2 \cdot S_k\right). \tag{9.6}$$

Wie bereits erwähnt, so gehen wir für unsere folgenden Überlegungen davon aus, dass dieser Fetch-On-Write-Overhead nicht vorhanden ist.

9.2.3 Regeln für die Kernelfusion

Als Nächstes werden wir diejenigen allgemeinen Regeln aufstellen, die unser Ansatz der Kernelfusion bei dem Verschmelzen von Vektorgrundoperationen befolgen muss. Diese Regeln setzen sich zusammen aus allgemeinen Voraussetzungen, die die Kernelfusion befolgen muss, und weitere von diesen Voraussetzungen und dem Speicherzugriffsmuster der Vektorgrundoperationen abgeleitete Regeln, wann eine Abhängigkeit zwischen zwei Vektorgrundoperationen zu einem Kernel verschmelzbar ist. In unserem Ansatz lauten die allgemeinen Voraussetzungen für die Kernelfusion wie folgt:

Block einer Vektor-Grundoperation	Gelesene Komponenten der Argumentvektoren	Geschriebene Komponenten des Ergebnisvektors
RHS	Beliebige Komponenten	Zugewiesene Komponenten
LC	Zugewiesene Komponenten	Zugewiesene Komponenten
MAP	Zugewiesene Komponenten	Zugewiesene Komponenten
RED	Zugewiesene Komponenten	n.a (blockweite Reduktion)

Tabelle 9.1: Speicherzugriffmuster für die Grundoperationen eines Blocks eines Kernels. Für diese Tabelle nehmen wir an, dass dem Block des Kernels eine beliebige Menge an Systemkomponenten zur Berechnung zugewiesen wurde.

i) Jede Vektorgrundoperation muss von genau einem Kernel berechnet werden.

ii) Es darf keine zyklische Abhängigkeit zwischen den ausführbaren Objekten, das heißt den Kernels, der Konstantengenerierung oder der Schrittweitenkontrolle, existieren, weil der Prozessor diese nacheinander berechnen muss.

iii) Es dürfen keine Abhängigkeiten zwischen den parallelen Blöcken eines Kernels existieren, weil ein Prozessor nicht alle Blöcke des Kernels zeitgleich ausführen kann.

Für die Verschmelzung von Vektorgrundoperationen zu einem fusionierten Kernel setzen wir voraus, dass für alle zu fusionierenden Vektorgrundoperationen das gleiche $n : 1$ Mapping zwischen deren Vektorkomponenten und den parallelen Blöcken des Kernels existiert. Dann zeigen die Blöcke des Kernels für die Vektorgrundoperationen folgendes Speicherzugriffsmuster, welches wir auch in Tabelle 9.1 zusammengefasst zeigen:

- **RHS-Operation:** Obwohl jeder Block auf Grund des beliebigen Zugriffsmusters der RHS-Funktion beliebige Komponenten des Eingabevektors lesen kann, so schreibt er nur seine zugewiesenen Komponenten in den Ergebnisvektor. Folglich muss der komplette Eingabevektor einer RHS-Operation vorlegen, bevor sie auch nur eine ihrer Komponenten berechnen kann.
- **(LC | MAP)-Operation:** Jeder Block liest nur seine zugewiesenen Komponenten aus den Eingabevektoren und schreibt nur seine zugewiesenen Komponenten in den Ergebnisvektor.
- **RED-Operation:** Hier liest jeder Block zunächst nur seine zugewiesenen Komponenten des Eingabevektors ein, um auf diesen dann eine partielle Reduktion durchzuführen und das Ergebnis hiervon über eine atomare Operation in einer globalen Variable abzuspeichern. Folglich ist das Ergebnis einer RED-Operation erst verfügbar, nachdem alle Blöcke die RED-Operation abgeschlossen haben.

Aus den allgemeinen Regeln für die Kernelfusion und dem Speicherzugriffsmuster der Vektorgrundoperationen können wir folgende Regeln für die Fusionierbarkeit zweier Vektorgrundoperationen aufstellen, indem wir zusätzlich den Typ der beiden Vektorgrundoperationen und die Art der Abhängigkeit zwischen den beiden Vektorgrundoperationen betrachten:

$a \to b$	b			
a	RHS	LC	MAP	RED
RHS		✓	✓	✓
LC	x	✓	✓	✓
MAP	x	✓	✓	✓
RED				

Tabelle 9.2: Fusionierbarkeit zweier Vektorgrundoperationen a und b bei einer direkten Abhängigkeit $a \to b$. Die fehlenden Einträge in dieser Tabelle symbolisieren, dass eine solche Abhängigkeit in einem Datenflussgraphen nicht auftreten kann (siehe Tabelle 6.1).

- **Keine Abhängigkeit:** Falls keine Abhängigkeit zwischen zwei Vektorgrundoperationen besteht, dürfen wir diese zu einem Kernel verschmelzen. Da es sich bei zwei voneinander unabhängigen Vektorgrundoperationen immer um Methodenparallelität handelt, ist das Verschmelzen von voneinander unabhängigen Vektorgrundoperationen auch ein Ausnutzen dieser Art von Parallelität.

- **Direkte Abhängigkeit:** Im Falle einer direkten Abhängigkeit zwischen zwei Vektorgrundoperationen gilt Folgendes (siehe Tabelle 9.2):

  - **(MAP | LC) $\to$ RHS-Abhängigkeiten (unfusionierbar):** Im allgemeinen Fall von beliebig gekoppelten Anfangswertproblemen benötigt die Auswertung einer Komponente der RHS-Operation Zugriff auf den kompletten Argumentvektor. Dadurch würde ein Block eines Kernels nach der Fusion für die RHS-Operation auch diejenigen Komponenten von dem Argumentvektor benötigen, die ein anderer Block des Kernels bei der Berechnung der (LC | MAP)-Operation produziert hätte. Aus diesem Grund würde die Fusion einer solchen (MAP | LC) $\to$ RHS-Abhängigkeit bei beliebig gekoppelten Anfangswertproblemen immer eine Abhängigkeit zwischen den Blöcken des Kernels verursachen, wodurch sie gemäß Voraussetzung iii unfusionierbar ist. Somit können wir (MAP | LC) $\to$ RHS-Abhängigkeiten bei beliebig gekoppelten Anfangswertproblemen als globale Barrieren betrachten.
  - **(RHS | LC | MAP) $\to$ (LC | MAP | RED)-Abhängigkeiten (fusionierbar):** Da bei solchen Abhängigkeiten und bei einem gleichen Mapping zwischen den Vektorkomponenten und den Blöcken des Kernels ein Block bei der Berechnung der (RHS | LC | MAP)-Operation am Anfang der Abhängigkeit genau diejenigen Komponenten erzeugt, welche der Block benötigt, um die zugewiesenen Komponenten von der (LC | MAP | RED)-Operation am Ende der Abhängigkeit zu berechnen, erzeugt diese Fusion keine Abhängigkeiten zwischen den Blöcken des Kernels. Somit sind (RHS | LC | MAP) $\to$ (LC | MAP | RED)-Abhängigkeiten gemäß Voraussetzung iii fusionierbar.

- **Indirekte Abhängigkeit:** Im Falle einer indirekten Abhängigkeit zwischen zwei Vektorgrundoperationen ist diese indirekte Abhängigkeit nur dann zu einem Kernel fusio-

nierbar, falls sie nur durch ein Netzwerk aus (LC | MAP)-Operationen getragen wird. Andernfalls, falls diese indirekte Abhängigkeit von einer (RHS | RED)-Operation oder von einer Schrittweitensteuerung mit Konstantengenerierung getragen wird, so ist diese nicht fusionierbar. Im Falle der Fusion einer indirekten Abhängigkeit müssen sämtliche Vektorgrundoperationen, die die Abhängigkeit tragen, ebenfalls fusioniert werden, da ansonsten eine zyklische Abhängigkeit zwischen zwei Kernels entstehen würde, welche gemäß der Regel ii verboten ist.

9.2.4 Speicheroptimierungen in einem Kernel

Als Nächstes werden wir im Detail vorstellen, wie ein fusioniertes Kernel den On-Chip-Speicher des Prozessors für die Datenwiederverwendung verwenden kann, um so die über den DRAM übertragene Datenmenge zu reduzieren:

Datentransfer für fusionierte (RHS | LC | MAP) $\rightarrow$ (LC | MAP | RED)-Abhängigkeiten über den On-Chip-Speicher des Prozessors (siehe Abbildung 9.1): Bei der Berechnung einer (RHS | MAP | LC)-Operation kann ein Block eines Kernels seine produzierten Komponenten in einen On-Chip-Speicher des Prozessors schreiben. Anschließend kann er diese Komponenten für die Berechnung einer davon direkt abhängigen (LC | MAP | RED)-Operation wieder aus dem On-Chip-Speicher auslesen. Dadurch findet der Datentransfer innerhalb eines Kernels zwischen den fusionierten Vektorgrundoperationen nur über die On-Chip-Speicher des Prozessors und nicht über den DRAM statt.

Wiederverwendung eines permanenten Vektors aus dem DRAM zwischen mehreren fusionierten Vektorgrundoperationen (siehe Abbildung 9.2): Diese Art der Datenwiederverwendung ist offensichtlich bei mehreren fusionierten (LC | MAP | RED)-Operationen, die sich einen oder mehrere permanente Argumentvektoren, die aber nicht durch das Kernel selbst produziert werden, und dadurch zunächst im DRAM liegen, teilen. Denn ein Block eines fusionierten Kernels greift für die Auswertung von jeder dieser Operationen auf die gleichen Komponenten des permanenten Argumentvektors zu. Hierdurch kann der Prozessor bei dem ersten Zugriff auf den permanenten Argumentvektor diese Komponenten von dem DRAM in einen On-Chip-Speicher laden, und alle darauf folgenden Zugriffe über den On-Chip-Speicher abwickeln. Insbesondere teilen sich voneinander unabhängige LC-Operationen, wie sie bei den unabhängigen Stufen eines ODE-Verfahrens auftreten, oft viele Argumentvektoren, wodurch die Fusion dieser voneinander unabhängigen LC-Operationen viele DRAM-Zugriffe einsparen kann. Allerdings kann diese Art der Datenwiederverwendung auch auftreten, falls sich eine RHS-Operation ihren permanenten Argumentvektor mit einer fusionierten (LC | MAP | RED)-Operation teilt. Dies setzt aber voraus, dass die RHS-Operation lokal auf diejenigen Komponenten des Argumentvektors zugreift, die der Block selbst oder ein benachbarter Block des Kernels, den der Prozessor zeitgleich oder wenig später berechnet, für die Berechnung seiner Komponenten der fusionierten (LC | MAP | RED)-Operation benötigt. Falls ein Block jedoch für die Auswertung der RHS-Operation nur auf weit entfernte Komponenten zugreift, reicht die Größe On-Chip-Speichers nicht aus, damit der Prozessor die Komponenten bei einem deutlich späteren Block für die Berechnung der (LC | MAP | RED)-Operation oder der RHS-Operation noch darinnen vorrätig halten kann.

Kein unnötiges Zurückschreiben von Ergebnissen von (RHS | LC | MAP)-Operationen von dem On-Chip-Speicher in permanente Vektoren beziehungsweise in den DRAM (siehe Abbildung 9.3): Ein Kernel muss das Ergebnis einer von ihm berechneten (RHS | LC |

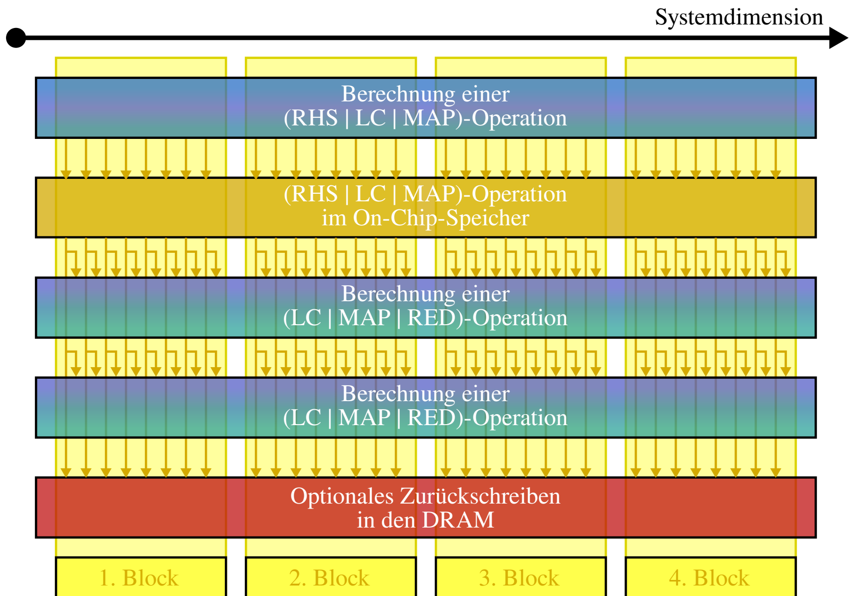


Abbildung 9.1: Datentransfer in einem fusionierten Kernel.

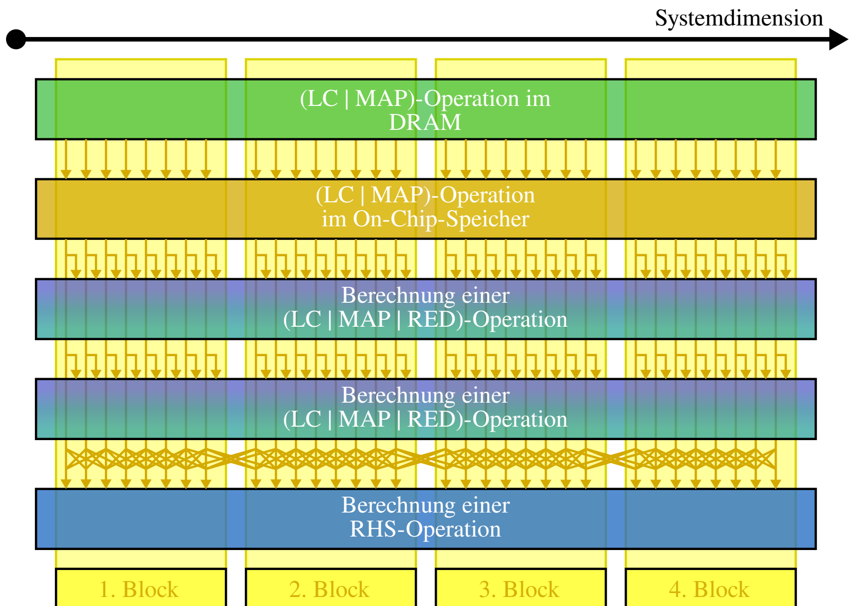
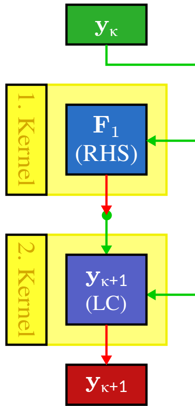


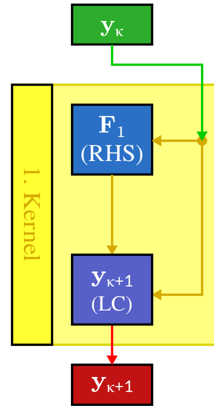
Abbildung 9.2: Vermeidung von redundanten Ladeoperationen in einem fusionierten Kernel.

Unfusioniertes Eulerverfahren



Gelesene Vektoren (⟶): 3
Geschriebene Vektoren (⟶): 2

Fusioniertes Eulerverfahren



Gelesene Vektoren (⟶): 1
Geschriebene Vektoren (⟶): 1

Abbildung 9.3: Reduktion der Anzahl an zurückgeschriebenen Vektoren durch Kernelfusion. Diese Abbildung zeigt anhand des expliziten Euler-Verfahrens, wie die Kernelfusion die Anzahl an in den DRAM zurückgeschriebenen Vektoren reduzieren kann. In der unfusionierten Variante der Basisimplementierung muss das 1. Kernel $\mathbf{F}_1$ in den DRAM zurückschreiben, da das 2. Kernel diesen Vektor zur Berechnung von $\mathbf{y}_{\kappa+1}$ benötigt. In der fusionierten Variante ist kein Zurückschreiben von $\mathbf{F}_1$ mehr nötig, da dieser Vektor nur vom fusionierten Kernel gelesen wird.

MAP)-Operation nur von dem On-Chip-Speicher in einen permanenten Vektor und damit in den DRAM zurückschreiben, falls dieses Ergebnis beziehungsweise der permanente Vektor von einem späteren Kernel gelesen wird. Falls jedoch das Ergebnis einer (RHS | LC | MAP)-Operation nicht von einem späteren Kernel gelesen wird, so ist kein Zurückschreiben in den DRAM erforderlich, wodurch die Kernelfusion wiederum DRAM-Zugriffe einsparen kann.

Dadurch muss der Prozessor bei einer Variante mit Kernelfusion, wenn wir zudem ein perfektes Caching annehmen, nur Daten zwischen unterschiedlichen Kernelaufrufen über den DRAM transferieren, während der Datentransfer innerhalb eines Kernels nur über den On-Chip-Speicher stattfindet. Ein durch die Kernelfusion partitionierter Datenflussgraph repräsentiert diesen Datenfluss über diejenigen Kanten, die die Kernels untereinander verbinden.

9.2.5 Mögliche Fusionen in einer Abhängigkeitskette

Bei ODE-Verfahren, die eine Abhängigkeitskette von RHS $\rightarrow$ LC-Gliedern besitzen, erlauben es die Regeln der Kernelfusion jeweils alle Vektorgrundoperationen in einem RHS $\rightarrow$ LC-Glied zu einem einzigen Kernel zu verschmelzen. Jedoch verbieten die Regeln es auch, die Vektorgrundoperationen aus unterschiedlichen RHS $\rightarrow$ LC-Gliedern zu fusionieren, da dies die Fusion einer LC $\rightarrow$ RHS-Abhängigkeit erfordern würde. Folglich spaltet die Kernelfusion bei

expliziten Einschrittverfahren eine Stufe typischerweise auf zwei aufeinanderfolgende Kernel auf: Das erste Kernel berechnet die LC-Operation der Stufe, während das zweite Kernel die RHS-Operation der Stufe berechnet. Dieses Aufspalten der Stufen bei der Fusion der RHS $\rightarrow$ LC-Abhängigkeitskette wird beispielhaft in Abbildung 9.4 für ein RK-Verfahren gezeigt.

Falls in einem ODE-Verfahren jedoch mehrere Stufen unabhängig voneinander sind, dann beinhaltet das entsprechende erste RHS $\rightarrow$ LC-Glied mehrere voneinander unabhängige LC-Operationen, und das darauf folgende RHS $\rightarrow$ LC-Glied mehrere unabhängige RHS-Operationen. Diese zusätzlichen LC-Operationen und RHS-Operationen können jeweils mit dem Kernel des entsprechenden Glieds verschmolzen werden. Dies wiederum bewirkt, dass die Kernels mehrere LC-Operationen und RHS-Operationen berechnen. Da LC-Operationen von unabhängigen Stufen typischerweise viele Argumentvektoren miteinander teilen, kann ein Prozessor so viele Daten über den On-Chip-Speicher wiederverwenden. Da unabhängige Stufen auch immer Methodenparallelität in einem ODE-Verfahren darstellen, können wir so über die Kernelfusion ebendiese Methodenparallelität ausnutzen, um die Datenwiederverwendung zu erhöhen.

9.2.6 Enabling-Transformationen

Bei *Enabling-Transformationen* (siehe zum Beispiel [B15]) handelt es sich um Programmtransformationen, das heißt Enabling-Transformationen modifizieren den Kontrollfluss oder den Datenfluss eines Programms. Dabei ist es nicht das Ziel dieser Enabling-Transformationen durch diese Modifikation direkt die Performance eines Programms zu erhöhen, sondern sie sollen nur weitere Programmtransformationen, welche dann letztendlich die Performance des Programms verbessern, ermöglichen. Wir wenden solche Enabling-Transformationen auf den Datenflussgraphen vor dem Fusionieren an, um die Datenwiederverwendung weiter zu erhöhen und darüber die Performance weiter zu verbessern. Diese Enabling-Transformationen verändern die Grundoperationen und deren Abhängigkeiten im Datenflussgraphen so, dass sich zwar das Ergebnis eines Zeitschritts nicht verändert, aber die Kernelfusion das Datenvolumen der erzeugten Kernels weiter reduzieren kann.

Für diesen Zweck betrachten wir in dieser Arbeit drei Enabling-Transformationen: Eine *Cloning-Transformation*, welche das redundante Auswerten einer Grundoperation von mehreren Kernels erlaubt, eine *Splitting-Transformation*, welche die Auswertung einer (LC | MAP)-Operationen auf mehrere Kernels verteilen kann, und eine *Move-Transformation*, welche Grundoperationen zwischen den Zeitschritten verschiebt. Das grundlegende Konzept der Cloning-Transformation wurde bereits in der Arbeit [Z23] verwendet, indem diese Arbeit bei einer händisch erstellten Implementierungsvariante für PIRK-Verfahren eine RHS-Operation redundanterweise mehrmals berechnen ließ. Ebenso kann die Cloning-Transformation in Kombination mit der darauffolgenden Kernelfusion als eine Variation der Optimierungstechnik der *Rematerialization* (siehe zum Beispiel [B15] oder [Z42]) betrachtet werden. Durch diese Technik der Rematerialization werden Werte, die von einem Programm bereits berechnet wurden und die nicht mehr in den Registern, sondern nur noch im Arbeitsspeicher vorliegen, falls möglich, aus anderen in den Registern vorliegenden Werten rekonstruiert, wodurch das Laden aus dem Arbeitsspeicher entfällt. Des Weiteren wurde das grundlegende Konzept der Splitting-Transformation bereits in der Arbeit [Z22] händisch auf CPU-Implementierungen für RK-Verfahren angewendet, so dass die resultierende Schleifenstruktur nach der Berechnung der RHS-Operation einer Stufe die Argumentvektoren der RHS-Operationen für alle folgenden Stufen um das soeben berechnete Ergebnis inkrementiert.

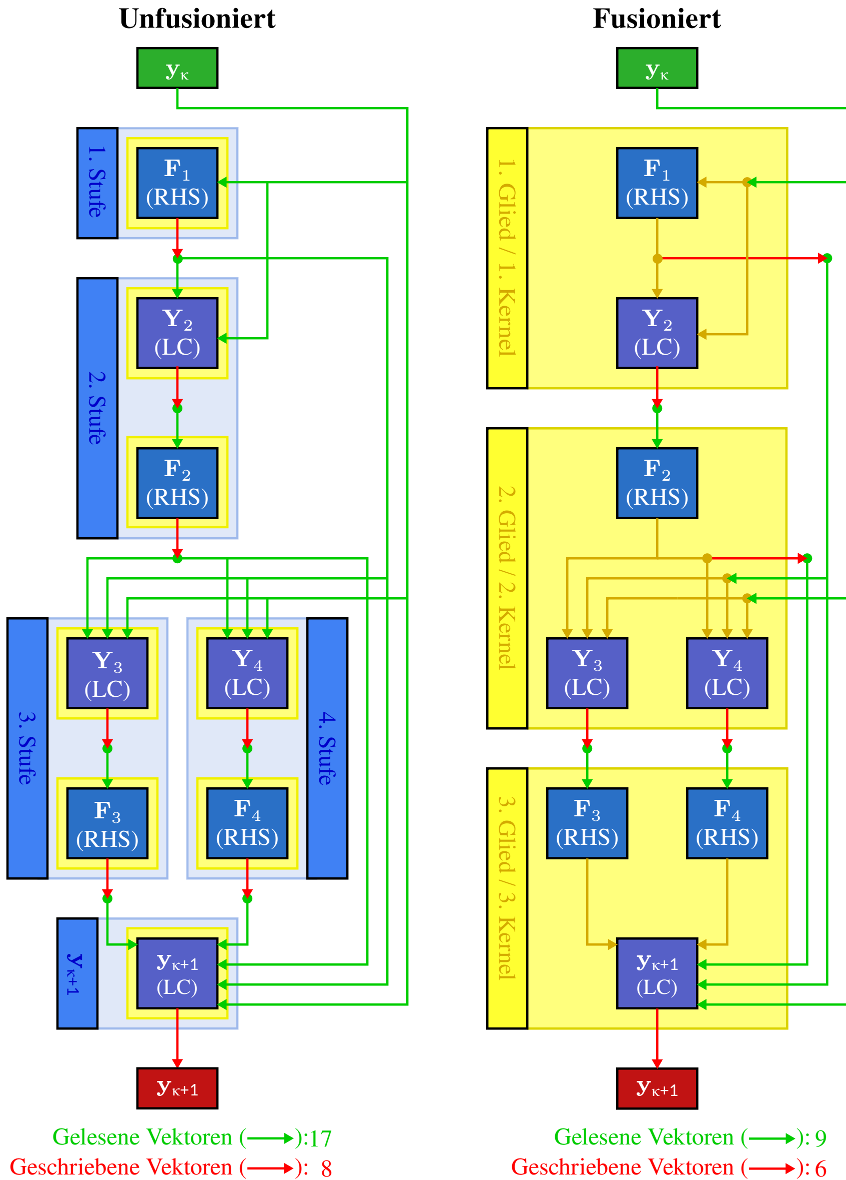


Abbildung 9.4: Mögliche Fusionen in der Abhängigkeitskette. Diese Abbildung zeigt beispielhaft die möglichen Fusionen in der Abhängigkeitskette eines vierstufigen RK-Verfahrens, bei welchem die dritte und vierte Stufe voneinander unabhängig sind.

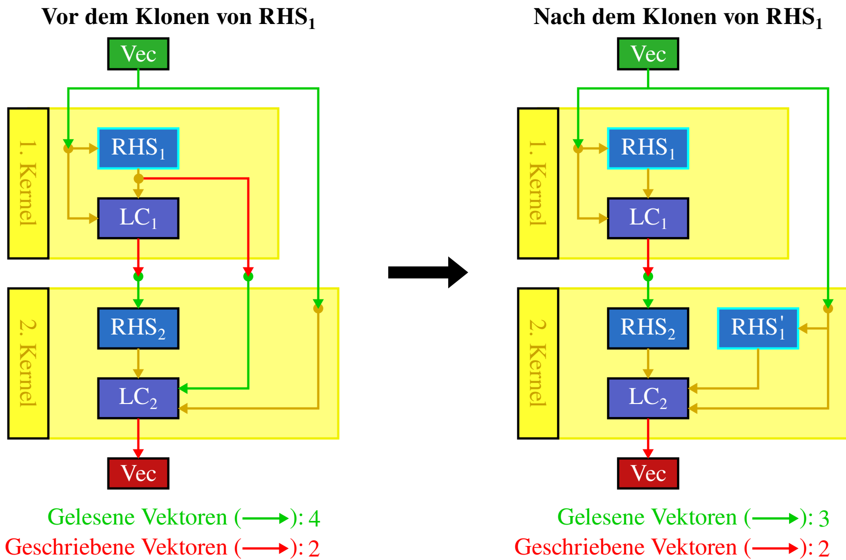


Abbildung 9.5: Cloning-Transformation. Diese Abbildung zeigt beispielhaft, wie die Cloning-Transformation die Anzahl an gelesenen und geschriebenen Vektoren reduzieren kann. In diesem Beispiel klonen wir den Knoten RHS_1 nach RHS'_1, so dass wir diesen Klon zum 2. Kernel hinzufügen können. Dadurch muss das Ergebnis von RHS_1 nicht von dem 1. Kernel in den DRAM zurückgeschrieben und anschließend von dem 2. Kernel geladen werden.

Cloning-Transformation (siehe Abbildung 9.5): Für die *Cloning-Transformation* nehmen wir zunächst an, dass ein Kernel k_2 bereits alle permanenten Argumentvektoren einer unverschmolzenen Vektorgrundoperation a lädt, um andere Vektorgrundoperationen zu berechnen. Folglich würde ein Hinzufügen von a zu dem Kernel k_2 das Datenvolumen dieses Kernels reduzieren. Jedoch können bei einem konkreten ODE-Verfahren die Abhängigkeiten der Vektorgrundoperation a zu anderen Grundoperationen ein Verschmelzen von a mit dem Kernel k_2 verhindern, wodurch a von dem früheren Kernel k_1 berechnet werden muss. In diesem Fall können wir im Datenflussgraphen den Knoten der Grundoperation a via der Cloning-Transformation zu einem Knoten a' klonen. Dabei hat das Klon a' den gleichen eingehenden Datenfluss und führt die gleichen Berechnungen wie die ursprüngliche Vektorgrundoperation a durch. Jedoch teilt die Cloning-Transformation den ausgehenden Datenfluss so zwischen a' und a auf, dass wir das Klon a' zu dem Kernel k_2 hinzufügen können. Dadurch muss das Kernel k_2 den Vektor a nicht mehr aus dem DRAM laden, und zusätzlich, falls dieser Vektor nur vom Kernel k_2 gelesen wurde, muss ihn das Kernel k_1 nicht mehr zurückschreiben. Somit verringert die Cloning-Transformation das Datenvolumen der Kernel k_1 und k_2. Dies geschieht aber auf Kosten eines arithmetischen Overheads, da die Kernel k_1 und k_2 die Vektorgrundoperation a redundanterweise je einmal berechnen.

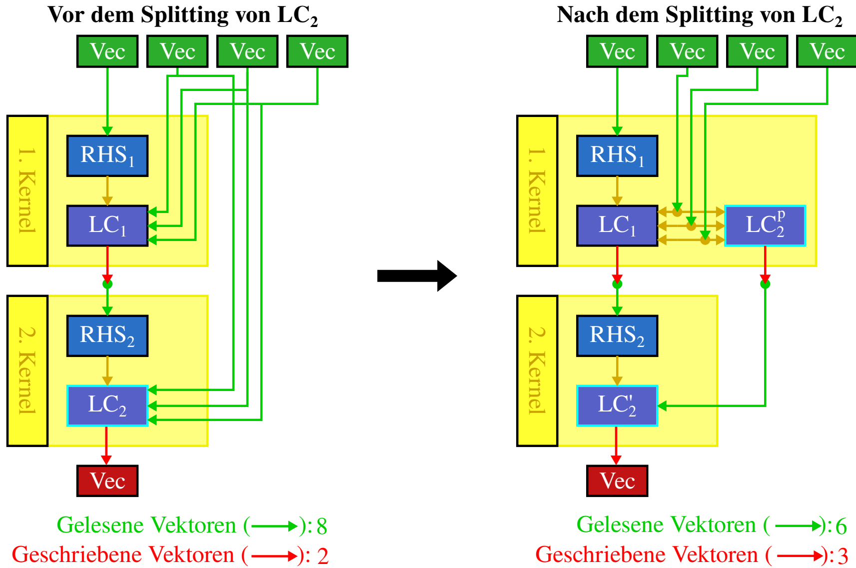


Abbildung 9.6: Splitting-Transformation. Diese Abbildung zeigt die Reduktion der Anzahl der aus dem DRAM gelesenen Vektoren durch die Splitting-Transformation. In diesem Beispiel spaltet die Splitting-Transformation die LC-Operation LC_2 so auf, dass das 1. Kernel diejenigen drei Argumentvektoren, die es bereits für LC_1 lädt, verwenden kann, um mit LC_2^p einen Teil von LC_2 zu berechnen. Diese partielle Linearkombination LC_2^p schreibt das 1. Kernel anschließend zurück, damit sie von dem 2. Kernel geladen und als LC_2' vervollständigt werden kann. Somit reduziert die Splitting-Transformation in diesem Beispiel die Anzahl der gelesenen Vektoren um zwei, erhöht aber die Anzahl der geschriebenen Vektoren um eins.

Splitting-Transformation (siehe Abbildung 9.6): Die *Splitting-Transformation* nützt aus, dass alle LC-Operationen und viele MAP-Operationen sich in mehrere partielle Operationen aufspalten lassen, wobei jede partielle Operation wieder eine (LC | MAP)-Operation ist und weniger Argumentvektoren als die ursprüngliche Operation benötigt (zum Beispiel $g(x_{1,j}, x_{2,j}, x_{3,j}, x_{4,j}) = g'(g^p(x_{1,j}, x_{2,j}, x_{3,j}), x_{4,j})$). Für die Erklärung der Splitting-Transformation nehmen wir an, dass ein erstes Kernel k_1 bereits mehrere Argumentvektoren einer aufspaltbaren (LC | MAP)-Operation lädt, aber noch nicht alle Argumentvektoren dieser Operation im Kernel k_1 verfügbar sind. Dadurch können wir diese (LC | MAP)-Operation selbst nicht zu dem Kernel k_1 hinzufügen. Allerdings kann die Splitting-Transformation diese in zwei partielle Operationen so aufspalten, dass das Kernel k_1 mit den bereits geladenen Vektoren möglichst viel von der aufgespaltenen Operation berechnet, und ein folgendes zweites Kernel k_2 die Operation vervollständigt. Dadurch muss das zweite Kernel k_2 diejenigen Argumentvektoren, die bereits von der partiellen Operation verarbeitet wurden, nicht mehr laden, sofern sie nicht von einer anderen Vektorgrundoperation in k_2 benötigt werden. Jedoch verursacht die Splitting-Transformation auch einen Overhead, da das erste Kernel k_1 das partielle Ergebnis der aufgespaltenen (LC | MAP)-Operation zunächst in den DRAM zurückschreiben und das zweite Kernel k_2 dieses partielle Ergebnis zur Vervollständigung aus dem DRAM laden muss.

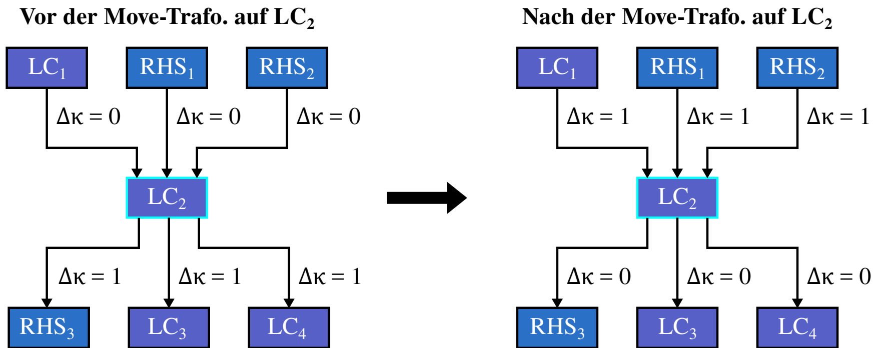


Abbildung 9.7: Move-Transformation. Diese Abbildung zeigt beispielhaft die Änderungen der Zeitschrittdistanzen bei der Anwendung der Move-Transformation. In diesem Beispiel wenden wir die Move-Transformation auf die LC-Operation LC_2 an, wodurch sich die Zeitschrittdistanzen der Argumentvektoren von LC_2 um jeweils eins erhöhen, während sich die Zeitschrittdistanzen zu denjenigen Grundoperationen, die das Ergebnis der von LC_2 lesen, um jeweils eins verringern.

Move-Transformation (siehe Abbildung 9.7): Die *Move-Transformation* verschiebt den Knoten beziehungsweise die Auswertung einer Grundoperation von dem aktuellen Zeitschritt in den nächsten Zeitschritt. Hierfür erhöht die Move-Transformation die Zeitschrittdistanzen der eingehenden Kanten um jeweils eins, während sie die Zeitschrittdistanzen der ausgehenden Kanten um jeweils eins verringert. Offensichtlich dürfen beim Verschieben einer Grundoperation keine Abhängigkeiten von einem zukünftigen zum aktuellen Zeitschritt entstehen. Aus diesem Grund dürfen nach den Move-Transformationen keine Zeitschrittdistanzen kleiner als null auftreten. Gegebenenfalls können wir aber die Move-Transformation mehrmals auf unterschiedliche Vektorgrundoperationen anwenden, um solche invaliden Zeitschrittdistanzen zu vermeiden. Zudem darf die Move-Transformation nicht den Knoten der Schrittweitensteuerung verschieben, da diese am Ende des aktuellen Zeitschritts ausgeführt werden muss.

9.2.7 Anwendung der Splitting-Transformation auf eine Abhängigkeitskette

Da in ODE-Verfahren mit einer längeren RHS $\rightarrow$ LC-Abhängigkeitskette, wie zum Beispiel RK-Verfahren, ein (RHS | LC)-Vektor oft in vielen folgenden Gliedern der Abhängigkeitskette von einer LC-Operation gelesen wird, ist ein Aufspalten von LC-Operationen mit Hilfe der Splitting-Transformation meist vorteilhaft. Für die Anwendung der Splitting-Transformation auf die Abhängigkeitskette müssen wir allerdings Folgendes beachten:

- Die Splitting-Transformation erzeugt einen Overhead, da durch sie zunächst ein Kernel ein partielles Ergebnis einer LC-Operation in den DRAM schreiben und danach ein darauffolgendes Kernel das partielle Ergebnis wieder lesen muss. Dadurch erzeugt ein maximales Aufspalten der LC-Operationen in der Abhängigkeitskette ein sehr hohes Datenvolumen.

- Beinhaltet ein RHS $\rightarrow$ LC-Glied der Abhängigkeitskette mehrere LC-Operationen, die sich mehrere permanente Argumentvektoren aus dem DRAM teilen, so muss die Splitting-Transformation all diese LC-Operationen aufspalten, damit das Kernel des Glieds die von den LC-Operationen abgespaltenen permanenten Argumentvektoren nicht mehr aus dem DRAM laden muss.
- Es ist oft vorteilhaft per Splitting-Transformation mehrere partielle LC-Operationen in einem Kernel zu berechnen, da dann die folgenden Kernel nur noch die Ergebnisse der partiellen LC-Operationen und nicht mehr deren Argumentvektoren aus dem DRAM laden müssen.
- Es ist mit der Splitting-Transformation möglich, das Zurückschreiben eines (RHS | LC)-Vektors in den DRAM zu vermeiden, wenn sämtliche LC-Operationen, die diesen Vektor lesen, so aufgespalten werden, dass die Fusion die partiellen LC-Operationen mit dem Kernel des Glieds, welches diesen (RHS | LC)-Vektor produziert, verschmelzen kann.

Da diese Splitting-Transformationen nicht nur einen Overhead erzeugen, sondern auch untereinander interagieren, gibt es wahrscheinlich keinen einfachen Algorithmus, welcher das Datenvolumen per Anwendung der Splitting-Transformation auf einer RHS $\rightarrow$ LC-Abhängigkeitskette minimiert, und welcher zudem nicht auf einer Suche mit einer Heuristik basiert.

9.2.8 Anwendung der Kernelfusion auf mehrere ODE-Verfahren

Im Folgenden werden wir per Kernelfusion und Enabling-Transformationen mehrere Implementierungsvarianten von den expliziten RK-Verfahren, den PIRK-Verfahren und den Peer-Verfahren mit einem minimierten Datenvolumen ableiten:

- **RK-Verfahren:** Um für ein RK-Verfahren per Kernelfusion eine Implementierungsvariante mit einem minimierten Datenvolumen zu generieren, müssen wir spezifisch auf dessen Struktur beziehungsweise die Besetztheit von dessen Koeffizienten A, $\mathbf{b}$ und $\hat{\mathbf{b}}$ eingehen, um so die unabhängigen Stufen zu erkennen und zu fusionieren, sowie um die Splitting-Transformation effizient anzuwenden. Aus diesem Grund ist es nicht möglich per Kernelfusion allgemeine optimierte Implementierungsvarianten für die Klasse der RK-Verfahren zu generieren, weshalb wir im Folgenden nur optimierte Implementierungsvarianten für zwei ausgewählte RK-Verfahren, und zwar das Verner-Verfahren und das DOPRI-Verfahren, vorstellen.
- **PIRK-Verfahren:** Da im Gegensatz zu den allgemeineren RK-Verfahren sämtliche PIRK-Verfahren, unabhängig von der Besetzung der Koeffizienten A und $\mathbf{b}$ des Basisverfahrens, eine sehr ähnliche Berechnungsstruktur besitzen, und es bei den PIRK-Verfahren nicht möglich ist, die Splitting-Transformation sinnvoll anzuwenden, können wir für die Klasse der PIRK-Verfahren allgemein Implementierungsvarianten mit einem minimierten Datenvolumen erstellen.
- **Peer-Verfahren:** Ebenso besitzen sämtliche Peer-Verfahren, unabhängig von der Besetzung ihrer Koeffizienten A und B, eine sehr ähnliche Berechnungsstruktur, und zudem ist es ebenfalls nicht möglich, die Splitting-Transformation für die Peer-Verfahren sinnvoll anzuwenden. Deshalb können wir auch allgemein für die Klasse der Peer-Verfahren Implementierungsvarianten mit einem minimierten Datenvolumen vorstellen.

9.2.8.1 Verner-Verfahren

Als Erstes betrachten wir die Herleitung der fusionierten Implementierungsvarianten des Verner-Verfahrens, ein eingebettetes, explizites RK-Verfahren mit einer Ordnung 5(6). Das Verner-Verfahren ist interessant für unsere Untersuchungen, da es eine große Anzahl von $s = 8$ Stufen besitzt und die sechste und die siebte Stufe voneinander unabhängig sind. Des Weiteren stehen im Butcher-Tableau des Verner-Verfahrens mehrere Nulleinträge unterhalb der Diagonalen. Das Butcher-Tableu dieses Verfahrens wird im Abschnitt 13.7 gezeigt. Wir erstellten drei unterschiedliche Implementierungsvarianten für das Verner-Verfahren, die wir noch einmal im Detail in den Tabellen 9.3 und 9.4 zeigen:

- **Verner-BAS (Basisimplementierung):** *Verner-BAS* ist eine unfusionierte Basisimplementierung, welche je ein Kernel pro Vektorgrundoperation startet. Somit muss Verner-BAS für die acht Stufen des Verner-Verfahrens acht LC-Kernel und acht RHS-Kernel starten. Abschließend startet Verner-BAS noch zwei LC-Kernel für $\mathbf{y}_{\kappa+1}$ und $\mathbf{E}$ sowie ein RED-Kernel für ϵ.

- **Verner-FUS (Nur Fusion):** *Verner-FUS* verschmilzt so viele Vektorgrundoperationen wie möglich, verwendet aber keine Enabling-Transformationen. Folglich verschmilzt Verner-FUS jeweils ein RHS $\rightarrow$ LC-Glied der Abhängigkeitskette des Verner-Verfahrens zu einem Kernel, wobei es wegen der beiden unabhängigen Stufen insgesamt 7 RHS $\rightarrow$ LC-Glieder und damit 7 Kernel gibt. Somit beinhaltet jedes Kernel von Verner-FUS eine RHS-Operation gefolgt von einer LC-Operation, wodurch in Verner-FUS jede Stufe des Verner-Verfahrens über zwei aufeinanderfolgende Kernel verteilt ist. Eine Ausnahme hiervon sind das 5. Kernel und das 6. Kernel, die jeweils einen Teil der beiden unabhängigen Stufen berechnen: Das 5. Kernel berechnet die LC-Operationen der 6. Stufe und 7. Stufe, während das 6. Kernel die RHS-Operationen der 6. Stufe und 7. Stufe berechnet.

- **Verner-FUSEN (Fusion und Enabling-Transformationen):** *Verner-FUSEN* wendet zusätzlich die Enabling-Transformationen vor dem Fusionieren an. Folglich, als eine Verbesserung zu Verner-FUS, kann nun auf Grund der Splitting-Transformation das 3. Kernel einen Teil der LC-Operation von $\mathbf{Y}_5$ berechnen, welcher dann vom 4. Kernel vervollständigt wird. Ebenso kann nun auf Grund der Splitting-Transformation das 5. Kernel einen Teil der LC-Operationen von $\mathbf{Y}_5$, $\mathbf{Y}_8$, $\mathbf{y}_{\kappa+1}$ und $\mathbf{E}$ berechnen, die dann durch das 6. Kernel und 7. Kernel vervollständigt werden. Des Weiteren wendet Verner-FUSEN mehrmals die Cloning-Transformation auf den Knoten der RHS-Operation von $\mathbf{F}_1$ an, so dass jedes Kernel $\mathbf{F}_1$ aus $\mathbf{y}_\kappa$ berechnet. Dadurch muss wiederum kein Kernel $\mathbf{F}_1$ in den DRAM schreiben oder aus dem DRAM laden.

Die Übersicht über das Datenvolumen der Varianten vom Verner-Verfahren in der Tabelle 9.5 zeigt, dass Verner-BAS als Variante der Basisimplementierung 73 Vektoren über den DRAM zwischen den Kernels transferieren muss, während Verner-FUS diese Anzahl mit Hilfe der Kernelfusion auf 49 Vektoren und Verner-FUSEN mit Hilfe der zusätzlichen Enabling-Transformationen auf 37 Vektoren reduziert. Somit besitzt Verner-FUSEN nur noch 51 % des Datenvolumens von Verner-BAS.

Verner-BAS

Kernel	Berechnungen	Geladene Daten	Geschriebene Daten
1	$\mathbf{F}_1$	$\mathbf{y}_\kappa$	$\mathbf{F}_1$
2	$\mathbf{Y}_2$	$\mathbf{y}_\kappa, \mathbf{F}_1$	$\mathbf{Y}_2$
3	$\mathbf{F}_2$	$\mathbf{Y}_2$	$\mathbf{F}_2$
4	$\mathbf{Y}_3$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_2$	$\mathbf{Y}_3$
5	$\mathbf{F}_3$	$\mathbf{Y}_3$	$\mathbf{F}_3$
6	$\mathbf{Y}_4$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3$	$\mathbf{Y}_4$
7	$\mathbf{F}_4$	$\mathbf{Y}_4$	$\mathbf{F}_4$
8	$\mathbf{Y}_5$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3, \mathbf{F}_4$	$\mathbf{Y}_5$
9	$\mathbf{F}_5$	$\mathbf{Y}_5$	$\mathbf{F}_5$
10	$\mathbf{Y}_6$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3, \mathbf{F}_4, \mathbf{F}_5$	$\mathbf{Y}_6$
11	$\mathbf{F}_6$	$\mathbf{Y}_6$	$\mathbf{F}_6$
12	$\mathbf{Y}_7$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3, \mathbf{F}_4, \mathbf{F}_5$	$\mathbf{Y}_7$
13	$\mathbf{F}_7$	$\mathbf{Y}_7$	$\mathbf{F}_7$
14	$\mathbf{Y}_8$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3, \mathbf{F}_4, \mathbf{F}_5, \mathbf{F}_7$	$\mathbf{Y}_8$
15	$\mathbf{F}_8$	$\mathbf{Y}_8$	$\mathbf{F}_8$
16	$\mathbf{y}_{\kappa+1}$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_3, \mathbf{F}_4, \mathbf{F}_5, \mathbf{F}_6$	$\mathbf{y}_{\kappa+1}$
17	$\mathbf{E}$	$\mathbf{y}_\kappa, \mathbf{y}_{\kappa+1}, \mathbf{F}_1, \mathbf{F}_3, \mathbf{F}_4, \mathbf{F}_5, \mathbf{F}_7, \mathbf{F}_8$	$\mathbf{E}$
18	ϵ	$\mathbf{E}$	ϵ

Verner-FUS

Kernel	Berechnungen	Geladene Daten	Geschriebene Daten
1	$\mathbf{F}_1, \mathbf{Y}_2$	$\mathbf{y}_\kappa$	$\mathbf{F}_1, \mathbf{Y}_2$
2	$\mathbf{F}_2, \mathbf{Y}_3$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{Y}_2$	$\mathbf{F}_2, \mathbf{Y}_3$
3	$\mathbf{F}_3, \mathbf{Y}_4$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_2, \mathbf{Y}_3$	$\mathbf{F}_3, \mathbf{Y}_4$
4	$\mathbf{F}_4, \mathbf{Y}_5$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3, \mathbf{Y}_4$	$\mathbf{F}_4, \mathbf{Y}_5$
5	$\mathbf{F}_5, \mathbf{Y}_6, \mathbf{Y}_7$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3, \mathbf{F}_4, \mathbf{Y}_5$	$\mathbf{F}_5, \mathbf{Y}_6, \mathbf{Y}_7$
6	$\mathbf{F}_6, \mathbf{F}_7, \mathbf{Y}_8, \mathbf{y}_{\kappa+1}$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3, \mathbf{F}_4, \mathbf{F}_5, \mathbf{Y}_6, \mathbf{Y}_7$	$\mathbf{F}_7, \mathbf{Y}_8, \mathbf{y}_{\kappa+1}$
7	$\mathbf{F}_8, \mathbf{E}, \epsilon$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_3, \mathbf{F}_4, \mathbf{F}_5, \mathbf{F}_7, \mathbf{Y}_8, \mathbf{y}_{\kappa+1}$	ϵ

Tabelle 9.3: Kernel der Implementierungsvarianten des Verner-Verfahrens.

Verner-FUSEN

Kernel	Berechnungen	Geladene Daten	Geschriebene Daten
1	$\mathbf{F}_1, \mathbf{Y}_2$	$\mathbf{y}_\kappa$	$\mathbf{Y}_2$
2	$\mathbf{F}_1, \mathbf{F}_2, \mathbf{Y}_3$	$\mathbf{y}_\kappa, \mathbf{Y}_2$	$\mathbf{F}_2, \mathbf{Y}_3$
3	$\mathbf{F}_1, \mathbf{F}_3, \mathbf{Y}_4, \mathbf{Y}_5^p$	$\mathbf{y}_\kappa, \mathbf{F}_2, \mathbf{Y}_3$	$\mathbf{F}_3, \mathbf{Y}_4, \mathbf{Y}_5^p$
4	$\mathbf{F}_4, \mathbf{Y}_5$	$\mathbf{Y}_5^p, \mathbf{Y}_4$	$\mathbf{F}_4, \mathbf{Y}_5$
5	$\mathbf{F}_1, \mathbf{F}_5, \mathbf{Y}_6, \mathbf{Y}_7, \mathbf{Y}_8^p, \mathbf{y}_{\kappa+1}^p, \mathbf{E}^p$	$\mathbf{y}_\kappa, \mathbf{F}_2, \mathbf{F}_3, \mathbf{F}_4, \mathbf{Y}_5$	$\mathbf{Y}_6, \mathbf{Y}_7, \mathbf{Y}_8^p, \mathbf{y}_{\kappa+1}^p, \mathbf{E}^p$
6	$\mathbf{F}_6, \mathbf{F}_7, \mathbf{Y}_8, \mathbf{y}_{\kappa+1}$	$\mathbf{Y}_8^p, \mathbf{y}_{\kappa+1}^p \mathbf{Y}_6, \mathbf{Y}_7$	$\mathbf{F}_7, \mathbf{Y}_8, \mathbf{y}_{\kappa+1}$
7	$\mathbf{F}_8, \mathbf{E}, \epsilon$	$\mathbf{E}^p, \mathbf{F}_7, \mathbf{Y}_8, \mathbf{y}_{\kappa+1}$	ϵ

^{p} Partielle Linearkombinationen.

Tabelle 9.4: Kernel der Implementierungsvarianten des Verner-Verfahrens (Fortsetzung).

Variante	#Gelesene Vektoren	#Geschriebene Vektoren	#Summe
Verner-BAS	56	17	73
Verner-FUS	35	14	49
Verner-FUSEN	21	16	37

Tabelle 9.5: Zahl der gelesenen und geschriebenen Vektoren für das Verner-Verfahren.

9.2.8.2 Verfahren von Dormand und Prince

Als ein zweites eingebettetes, explizites RK-Verfahren betrachten wir nun das Verfahren von Dormand und Prince mit der Ordnung 5(4), welches wir im Folgenden als *DOPRI* abkürzen. Während dieses Verfahren weniger Potential für Optimierungen als das Verner-Verfahren bietet, so ist es weiter verbreitet und wird öfter zum Lösen von Anfangswertproblemen verwendet. DOPRI hat $s = 7$ Stufen mit der FSAL-Eigenschaft. Das Butcher-Tableu dieses Verfahrens wird im Abschnitt 13.7 gezeigt. Für das DOPRI-Verfahren betrachten wir mehrere Implementierungsvarianten, von denen wir *DOPRI-BAS*, *DOPRI-FUS* sowie *DOPRI-FUSEN* selbst erstellten, während *DOPRI-ODEINT* eine Variante aus der ODEINT-Bibliothek [W4] ist (siehe die Tabellen 9.7 und 9.6 für einen Überblick über die Kernelstruktur der Varianten):

- **DOPRI-BAS (Basisimplementierung):** *DOPRI-BAS* ist die unfusionierte Basisimplementierung, die die FSAL-Eigenschaft des Verfahrens ausnutzt, um sich eine LC-Operation und eine RHS-Operation einzusparen.
- **DOPRI-FUS (Nur Fusion):** *DOPRI-FUS* verschmilzt die RHS $\rightarrow$ LC-Glieder der Abhängigkeitskette zu jeweils einem Kernel, berücksichtigt dabei aber die FSAL-Eigenschaft, um sich die Berechnung einer RHS-Operation pro Zeitschritt einzusparen. Zudem verwendet DOPRI-FUS keine Enabling-Transformationen.
- **DOPRI-FUSEN (Fusion und Enabling-Transformationen):** *DOPRI-FUSEN* verschmilzt genauso wie DOPRI-FUS die RHS $\rightarrow$ LC-Glieder der Abhängigkeitskette zu jeweils einem Kernel. Jedoch, anstatt die FSAL-Eigenschaft auszunutzen, um sich eine RHS-Operation pro Zeitschritt einzusparen, erhöht DOPRI-FUSEN die Anzahl der RHS-Operationen sogar, indem es die Cloning-Transformation anwendet, um in jedem Kernel $\mathbf{F}_1$ aus $\mathbf{y}_\kappa$ zu rekonstruieren. Des Weiteren verwendet DOPRI-FUSEN die Splitting-Transformation, um im 2. Kernel und im 4. Kernel ein paar LC-Operationen partiell zu berechnen.
- **DOPRI-ODEINT (Variante aus der ODEINT-Bibliothek):** ODEINT [W4] stellt seinen Benutzer eine unfusionierte Implementierungsvariante des DOPRI-Verfahrens für GPUs und CPUs zur Verfügung, die wir in dieser Arbeit als *DOPRI-ODEINT* bezeichnen. DOPRI-ODEINT startet, ähnlich wie DOPRI-BAS, pro Vektorgrundoperation ein Kernel und nutzt genauso wie DOPRI-BAS die FSAL-Eigenschaft des Verfahrens aus. Jedoch berechnet DOPRI-ODEINT den Fehlervektor über $\mathbf{E} = h_\kappa \sum_{j=1}^{s}(\hat{b}_j - b_j)\mathbf{F}_j$ (Gleichung 4.13), während DOPRI-BAS den Fehlervektor über $\mathbf{E} = \mathbf{y}_\kappa - \mathbf{y}_{\kappa+1}$ (Gleichung 4.12) berechnet. Zuletzt verwendet DOPRI-ODEINT am Ende eines akzeptierten Zeitschritts kein Pointer-Swapping, um $\mathbf{y}_{\kappa+1}$ mit $\mathbf{y}_\kappa$ und $\mathbf{F}_7$ mit $\mathbf{F}_1$ zu ersetzen, sondern es kopiert $\mathbf{y}_{\kappa+1}$ nach $\mathbf{y}_\kappa$ sowie $\mathbf{F}_7$ nach $\mathbf{F}_1$. Für RHS-Operationen führt DOPRI-ODEINT einen Call-Back zu einer benutzerdefinierten Host-Funktion durch. Wir implementierten diese Host-Funktion so, dass sie ein RHS-Kernel auf dem Zielrechner startet.

Wir zeigen das Datenvolumen unserer drei Varianten und das Datenvolumen von DOPRI-ODEINT in der Tabelle 9.8. Der Vergleich des Datenvolumens zeigt auf, dass DOPRI-BAS als eine Variante der Basisimplementierung 54 Vektoren über den DRAM transferieren muss, während DOPRI-FUS diese Anzahl mit Hilfe der Kernelfusion auf 46 Vektoren und DOPRI-FUSEN mit Hilfe der zusätzlichen Enabling-Transformationen auf 32 Vektoren reduziert. Somit

DOPRI-BAS

Kernel	Berechnungen	Geladene Daten	Geschriebene Daten
1	$\mathbf{Y}_2$	$\mathbf{y}_\kappa, \mathbf{F}_1$	$\mathbf{Y}_2$
2	$\mathbf{F}_2$	$\mathbf{Y}_2$	$\mathbf{F}_2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
11	$\mathbf{y}_{\kappa+1}$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_3, \mathbf{F}_4, \mathbf{F}_5, \mathbf{F}_6$	$\mathbf{y}_{\kappa+1}$
12	$\mathbf{F}_7$	$\mathbf{y}_{\kappa+1}$	$\mathbf{F}_7$
13	$\mathbf{E}$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_3, \mathbf{F}_4, \mathbf{F}_5, \mathbf{F}_6,$ $\mathbf{F}_7, \mathbf{y}_{\kappa+1}$	$\mathbf{E}$
14	ϵ	$\mathbf{E}$	ϵ

DOPRI-FUS

Kernel	Berechnungen	Geladene Daten	Geschriebene Daten
1	$\mathbf{Y}_2$	$\mathbf{y}_\kappa, \mathbf{F}_1$	$\mathbf{Y}_2$
2	$\mathbf{F}_2, \mathbf{Y}_3$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{Y}_2$	$\mathbf{F}_2, \mathbf{Y}_3$
3	$\mathbf{F}_3, \mathbf{Y}_4$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_2, \mathbf{Y}_3$	$\mathbf{F}_3, \mathbf{Y}_4$
4	$\mathbf{F}_4, \mathbf{Y}_5$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3, \mathbf{Y}_4$	$\mathbf{F}_4, \mathbf{Y}_5$
5	$\mathbf{F}_5, \mathbf{Y}_6$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3, \mathbf{F}_4, \mathbf{Y}_5$	$\mathbf{F}_5, \mathbf{Y}_6$
6	$\mathbf{F}_6, \mathbf{y}_{\kappa+1}$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_3, \mathbf{F}_4, \mathbf{F}_5, \mathbf{F}_6, \mathbf{Y}_6$	$\mathbf{F}_6, \mathbf{y}_{\kappa+1}$
7	$\mathbf{F}_7, \mathbf{E}, \epsilon$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_3, \mathbf{F}_4, \mathbf{F}_5, \mathbf{F}_6, \mathbf{y}_{\kappa+1}$	$\mathbf{F}_7, \epsilon$

DOPRI-FUSEN

Kernel	Berechnungen	Geladene Daten	Geschriebene Daten
1	$\mathbf{F}_1, \mathbf{Y}_2$	$\mathbf{y}_\kappa$	$\mathbf{Y}_2$
2	$\mathbf{F}_1, \mathbf{F}_2, \mathbf{Y}_3, \mathbf{Y}_4^p$	$\mathbf{y}_\kappa, \mathbf{Y}_2$	$\mathbf{F}_2, \mathbf{Y}_3, \mathbf{Y}_4^p$
3	$\mathbf{F}_3, \mathbf{Y}_4$	$\mathbf{Y}_3, \mathbf{Y}_4^p$	$\mathbf{F}_3, \mathbf{Y}_4$
4	$\mathbf{F}_1, \mathbf{F}_4, \mathbf{Y}_5, \mathbf{Y}_6^p$ $\mathbf{y}_{\kappa+1}^p, \mathbf{E}^p$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_2, \mathbf{Y}_4$	$\mathbf{Y}_5, \mathbf{Y}_6^p, \mathbf{y}_{\kappa+1}^p, \mathbf{E}^p$
5	$\mathbf{F}_5, \mathbf{Y}_6$	$\mathbf{Y}_5, \mathbf{Y}_6^p$	$\mathbf{F}_5, \mathbf{Y}_6$
6	$\mathbf{F}_6, \mathbf{y}_{\kappa+1}$	$\mathbf{y}_{\kappa+1}^p, \mathbf{Y}_6, \mathbf{F}_5$	$\mathbf{F}_6, \mathbf{y}_{\kappa+1}$
7	$\mathbf{F}_7, \mathbf{E}, \epsilon$	$\mathbf{E}^p, \mathbf{F}_5, \mathbf{F}_6, \mathbf{y}_{\kappa+1}$	ϵ

Wegen der FSAL-Eigenschaft ist $\mathbf{y}_{\kappa+1}$ identisch mit $\mathbf{Y}_7$.
^{p} Partielle Linearkombination

Tabelle 9.6: Kernel der Implementierungsvarianten des DOPRI-Verfahrens.

DOPRI-ODEINT

Kernel	Berechnungen	Geladene Daten	Geschriebene Daten
1	$\mathbf{Y}_2$	$\mathbf{y}_\kappa, \mathbf{F}_1$	$\mathbf{Y}_2$
2	$\mathbf{F}_2$	$\mathbf{Y}_2$	$\mathbf{F}_2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
11	$\mathbf{y}_{\kappa+1}$	$\mathbf{y}_\kappa, \mathbf{F}_1, \mathbf{F}_3, \mathbf{F}_4, \mathbf{F}_5, \mathbf{F}_6$	$\mathbf{y}_{\kappa+1}$
12	$\mathbf{F}_7$	$\mathbf{y}_{\kappa+1}$	$\mathbf{F}_7$
13	$\mathbf{E}$	$\mathbf{F}_1, \mathbf{F}_3, \mathbf{F}_4, \mathbf{F}_5, \mathbf{F}_6, \mathbf{F}_7$	$\mathbf{E}$
14	ϵ	$\mathbf{E}$	ϵ
if $\epsilon < \epsilon_{\text{max}}$			
memcpy	n/a	$\mathbf{y}_{\kappa+1}$	$\mathbf{y}_\kappa$
memcpy	n/a	$\mathbf{F}_7$	$\mathbf{F}_1$

Tabelle 9.7: Kernel der DOPRI-ODEINT-Variante.

Variante	#Gelesene Vektoren	#Geschriebene Vektoren	#Summe
DOPRI-BAS	41	13	54
DOPRI-FUS	34	12	46
DOPRI-FUSEN	18	14	32
DOPRI-ODEINT	41/40	15/13	56/53

Tabelle 9.8: Zahl der gelesenen und geschriebenen Vektoren für das DOPRI-Verfahren. Bei DOPRI-ODEINT schwankt die Zahl der gelesenen und geschrieben Vektoren auf Grund des Kopierens, je nachdem ob ein Zeitschritt akzeptiert oder verworfen wurde.

besitzt DOPRI-FUSEN nur noch 59 % des Datenvolumens von DOPRI-BAS. DOPRI-ODEINT besitzt bei einem akzeptierten Zeitschritt mit 56 übertragenen Vektoren ein ähnlich großes Datenvolumen wie DOPRI-BAS mit 54 übertragenen Vektoren. Die Unterschiede erklären sich dadurch, dass beim DOPRI-Verfahren für dessen Belegung von $\mathbf{b}$ und $\hat{\mathbf{b}}$ die von DOPRI-ODEINT verwendete Berechnung des Fehlervektors per Gleichung 4.13 ein um zwei Vektoren geringeres Datenvolumen besitzt als die von DOPRI-BAS verwendete Berechnung per Gleichung 4.12. Allerdings muss DOPRI-ODEINT bei einem akzeptierten Zeitschritt zusätzlich die Vektoren $\mathbf{y}_{\kappa+1}$ und $\mathbf{F}_7$ am Ende eines Zeitschritts kopieren, wodurch ein zusätzliches Datenvolumen von vier Vektoren pro Zeitschritt anfällt.

9.2.8.3 PIRK-Verfahren

Als Nächstes wenden wir die Kernelfusion auf die PIRK-Verfahren aus Abschnitt 4.8 an, um so optimierte Implementierungsvarianten für diese zu erstellen. Da bei einem PIRK-Verfahren die Stufen eines Korrektorschritts unabhängig voneinander sind, sie nur von den RHS-Operationen des vorherigen Iterationschritts abhängen, und zudem sich die LC-Operationen eines Korrektorschritts sämtliche Argumentvektoren teilen, scheint die Fusion der unabhängigen Stufen eines Korrektorschritts bei PIRK-Verfahren vielversprechend zu sein, um das Datenvolumen stark zu reduzieren. Allerdings lässt sich wegen dieser Berechnungsstruktur auch weder die Splitting-Transformation noch die Cloning-Transformation auf die PIRK-Verfahren sinnvoll anwenden. In Anbetracht dieser Eigenschaften erstellten wir zwei Implementierungsvarianten (siehe Abbildung 9.8 für die Datenflussgraphen der Varianten eines beispielhaften PIRK-Verfahrens und Tabelle 9.9 für eine Übersicht über die Kernelstruktur der Varianten):

- **PIRK-BAS (Basisimplementierung):** *PIRK-BAS* startet als eine Variante der Basisimplementierung in jedem Korrektorschritt für jede Stufe des Basisverfahrens ein LC-Kernel und ein RHS-Kernel. Dies resultiert bei einem Basisverfahren mit s Stufen und m Korrektorschritten in insgesamt $2ms + 4$ ausgeführten Kernels.
- **PIRK-FUS (Nur Fusion):** *PIRK-FUS* fusioniert so viele Vektorgrundoperationen wie möglich. Folglich startet es zuerst ein Kernel für den Prädiktorschritt und anschließend ein weiteres Kernel für jeden Korrektorschritt, das heißt bei m Korrektorschritten insgesamt $m + 1$ Kernels. Das Kernel eines jeden Korrektorschritts berechnet die RHS-Operationen aller Stufen des aktuellen Korrektorschritts und die LC-Operationen des nächsten Korrektorschritts. Eine Ausnahme hiervon ist das Kernel des vorletzten und letzten Korrektorschritts: Das Kernel des vorletzten Korrektorschritts berechnet zusätzlich $\hat{\mathbf{y}}_{\kappa+1}$, und das Kernel des letzten Korrektorschritts berechnet zusätzlich $\mathbf{y}_{\kappa+1}$, $\mathbf{E}$ und ϵ. Da der Ergebnisvektor einer RHS-Operation nur von demjenigen Kernel benötigt wird, das diesen Vektor auch produziert, müssen die Kernels die Ergebnisvektoren der RHS-Operationen nicht in die permanenten Vektoren und damit in den DRAM zurückschreiben. Ebenso kann ein fusioniertes Kernel die Ergebnisvektoren der fusionierten RHS-Operationen eines Korrektorschritts über den On-Chip-Speicher wiederverwenden, um alle LC-Operationen des nächsten Korrektorschritts zu berechnen.

Die Tabelle 9.10 zeigt das Datenvolumen von PIRK-BAS und PIRK-FUS. Diese Tabelle verdeutlicht, dass das Datenvolumen bei PIRK-BAS eine quadratische Komplexität bezüglich der Stufenzahl des Basisverfahrens besitzt. Zum Beispiel müsste PIRK-BAS bei einem vollbesetzten Basisverfahren mit $s = 5$ Stufen und mit $m = 7$ durchgeführten Korrektorschritten 313 Vektoren über den DRAM transferieren. Dahingegen kann die Fusion bei PIRK-FUS das Datenvolumen auf eine lineare Komplexität bezüglich der Stufenzahl des Basisverfahrens reduzieren, wodurch PIRK-FUS für das soeben beispielhaft genannte PIRK-Verfahren nur noch 81 Vektoren über den DRAM transferieren müsste. Somit besitzt in diesem Beispiel PIRK-FUS nur noch 26 % des Datenvolumens von PIRK-BAS.

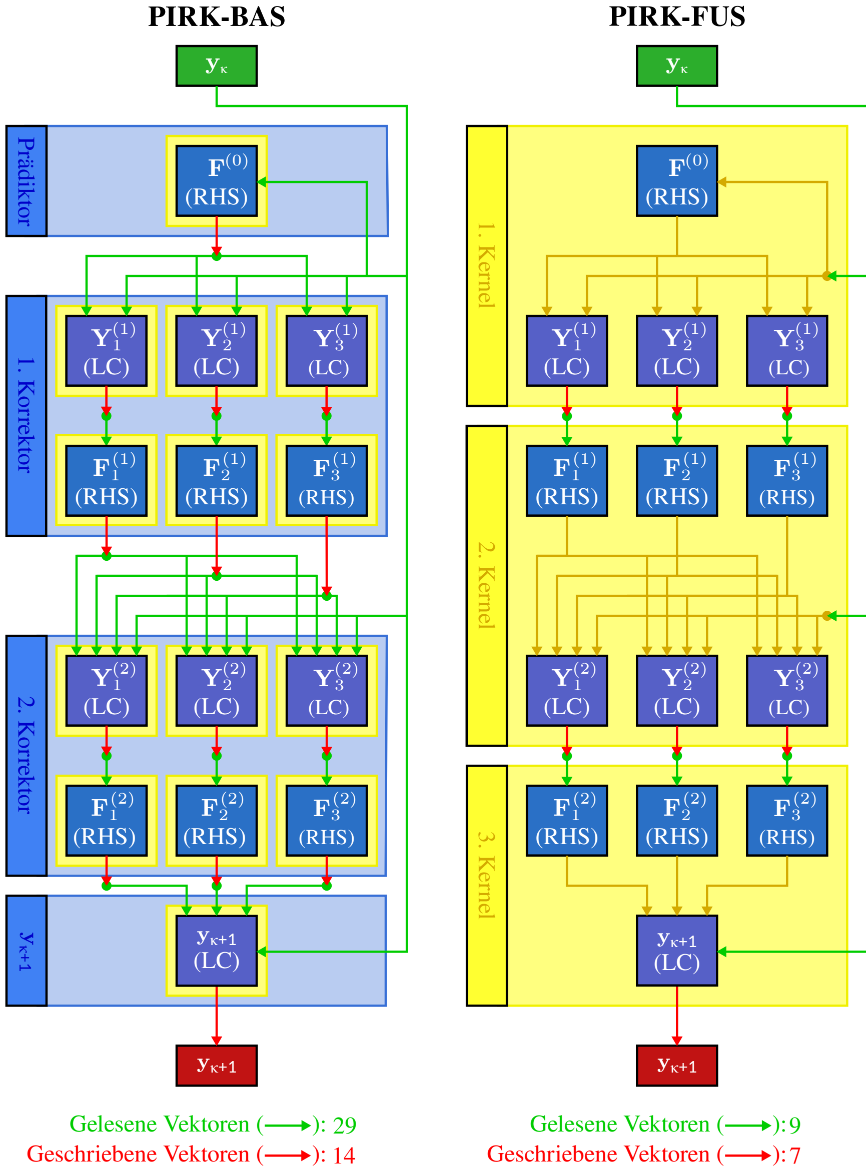


Abbildung 9.8: Datenflussgraphen der Varianten des PIRK-Verfahrens. Diese Abbildung zeigt beispielhaft die Datenflussgraphen der Varianten für ein PIRK-Verfahren mit einem vollbesetzten Basisverfahren mit drei Stufen und zwei Korrektorschritten. Die Berechnung des Fehlervektors $\mathbf{E}$ und lokalen Fehlers ϵ wird in diesem Beispiel nicht durchgeführt.

PIRK-BAS

Kernelstruktur	Berechnungen	Gelesene Daten	Geschr. Daten
1. Kernel	$\mathbf{F}_*^{(0)}$	$\mathbf{y}_\kappa$	$\mathbf{F}_*^{(0)}$
für jede Stufe $i = 1, \ldots, s$			
2. Kernel	$\mathbf{Y}_i^{(1)}$	$\mathbf{y}_\kappa, \mathbf{F}_*^{(0)}$	$\mathbf{Y}_i^{(1)}$
3. Kernel	$\mathbf{F}_i^{(1)}$	$\mathbf{Y}_i^{(1)}$	$\mathbf{F}_i^{(1)}$
für jeden Korr. $k = 2, \ldots, m$ für jede Stufe $i = 1, \ldots, s$			
4. Kernel	$\mathbf{Y}_i^{(k)}$	$\mathbf{y}_\kappa, \mathbf{F}_{1:s}^{(k-1)}$	$\mathbf{Y}_i^{(k)}$
5. Kernel	$\mathbf{F}_i^{(k)}$	$\mathbf{Y}_i^{(k)}$	$\mathbf{F}_i^{(k)}$
6. Kernel	$\mathbf{y}_{\kappa+1}$	$\mathbf{y}_\kappa, \mathbf{F}_{1:s}^{(m)}$	$\mathbf{y}_{\kappa+1}$
7. Kernel	$\mathbf{E}$	$\mathbf{y}_\kappa, \mathbf{y}_{\kappa+1}, \mathbf{F}_{1:s}^{(m-1)}$	$\mathbf{E}$
8. Kernel	ϵ	$\mathbf{E}$	ϵ

PIRK-FUS

Kernelstruktur	Berechnungen	Gelesene Daten	Geschr. Daten
1. Kernel	$\mathbf{F}_*^{(0)}, \mathbf{Y}_{1:s}^{(1)}$	$\mathbf{y}_\kappa$	$\mathbf{Y}_{1:s}^{(1)}$
für jeden Korr. $k = 1, \ldots, m-2$			
2. Kernel	$\mathbf{F}_{1:s}^{(k)}, \mathbf{Y}_{1:s}^{(k+1)}$	$\mathbf{y}_\kappa, \mathbf{Y}_{1:s}^{(k)}$	$\mathbf{Y}_{1:s}^{(k+1)}$
3. Kernel	$\mathbf{F}_{1:s}^{(m-1)}, \mathbf{Y}_{1:s}^{(m)}$, $\hat{\mathbf{y}}_{\kappa+1}$	$\mathbf{y}_\kappa, \mathbf{Y}_{1:s}^{(m-1)}$	$\mathbf{Y}_{1:s}^{(m)}$, $\hat{\mathbf{y}}_{\kappa+1}$
4. Kernel	$\mathbf{F}_{1:s}^{(m)}, \mathbf{y}_{\kappa+1}, \mathbf{E}, \epsilon$	$\mathbf{y}_\kappa, \mathbf{Y}_{1:s}^{(m)}, \hat{\mathbf{y}}_{\kappa+1}$	$\mathbf{y}_{\kappa+1}, \epsilon$

Tabelle 9.9: Kernel der Implementierungsvarianten der PIRK-Verfahren. Hierbei ist s die Anzahl an Stufen des Basisverfahrens und m die Anzahl an durchgeführten Korrektorschritten.

Variante	#Gelesene Vektoren	#Geschriebene Vektoren	#Summe
PIRK-BAS	$(m-1)s^2 + (2m+3)s + 5$	$2ms + 3$	$(m-1)s^2 + (4m+3)s + 8$
PIRK-FUS	$ms + m + 2$	$ms + 2$	$2ms + m + 4$

Tabelle 9.10: Zahl der gelesenen und geschriebenen Vektoren für das PIRK-Verfahren. Hierbei ist s wieder die Anzahl an Stufen des Basisverfahrens und m die Anzahl an durchgeführten Korrektorschritten.

9.2.8.4 Peer-Verfahren

Zuletzt betrachten wir die Anwendung der Kernelfusion und der Enabling-Transformationen auf die expliziten Peer-Verfahren aus Abschnitt 4.9. Ähnlich zu den PIRK-Verfahren besitzen die Peer-Verfahren unabhängige Stufen, die wir mit Hilfe von Kernelfusion verschmelzen können. Insgesamt erstellten wir mit Hilfe der Kernelfusion und den Enabling-Transformationen drei unterschiedliche Varianten für die Peer-Verfahren (siehe Abbildung 9.9 für eine Illustration der Varianten für ein zwei-stufiges Peer-Verfahren und Tabelle 9.11 für einen Überblick über die Kernelstruktur der Varianten):

- **Peer-BAS (Basisimplementierung):** *Peer-BAS* ist eine Variante der Basisimplementierung, die ein Kernel pro Vektorgrundoperation startet. Zuerst startet Peer-BAS für jede Stufe ein LC-Kernel, um den neuen Systemzustand der jeweiligen Stufe zu berechnen. Anschließend startet Peer-BAS für jede Stufe ein RHS-Kernel, um die neue zeitliche Ableitung der jeweiligen Stufe aus deren neuen Systemzustand zu ermitteln. Als Nächstes startet es ein MAP-Kernel, um den Fehlervektor zu bestimmen, und abschließend ein RED-Kernel, um den Fehlervektor zu dem lokalen Fehler zu reduzieren.
- **Peer-FUS (Nur Fusion):** *Peer-FUS* fusioniert so viele Vektorgrundoperationen zu einem einzigen Kernel wie möglich, verwendet aber keine Enabling-Transformationen. Folglich verschmilzt Peer-FUS die LC-Operationen aller Stufen zur Berechnung der neuen Systemzustände zu einem 1. Kernel. Zudem verschmilzt es die RHS-Operationen zur Berechnung der neuen zeitlichen Ableitungen der Stufen sowie die MAP-Operation zur Berechnung des Fehlervektors und die RED-Operation zur Berechnung des lokalen Fehlers zu einem 2. Kernel.
- **Peer-FUSEN (Fusion und Enabling-Transformationen):** *Peer-FUSEN* besitzt eine ähnliche Struktur wie Peer-FUS, wendet aber vor der Fusion die Cloning- und Move-Transformation auf die RHS-Operationen der Stufen an, und fusioniert die geklonten RHS-Operationen mit dem 1. Kernel. Dadurch kann das 1. Kernel von Peer-FUSEN die aktuellen Systemzustände der Stufen verwenden, um deren aktuellen zeitlichen Ableitungen zu rekonstruieren, und muss diese nicht aus dem DRAM laden. Infolgedessen schreibt das 2. Kernel von Peer-FUSEN die neuen zeitlichen Ableitungen auch nicht in den DRAM zurück.

Die Tabelle 9.12 mit dem Datenvolumen der Varianten der Peer-Verfahren zeigt, dass das Datenvolumen von Peer-BAS eine quadratische Komplexität bezüglich der Stufenzahl besitzt. Bei einem beispielhaften vierstufigen Peer-Verfahren ergibt sich so bei Peer-BAS ein Datenvolumen von 51 Vektoren. Durch die Anwendung der Kernelfusion lässt sich bei Peer-FUS diese quadratische Komplexität des Datenvolumens bezüglich der Stufenzahl auf eine lineare Komplexität reduzieren, wodurch bei dem soeben genannten vierstufigen Peer-Verfahren Peer-FUS nur noch ein Datenvolumen von 21 Vektoren besitzt. Die Enabling-Transformationen von Peer-FUSEN können das Datenvolumen im Vergleich zu Peer-FUS noch einmal um circa 40 % verringern, wodurch Peer-FUSEN bei dem vierstufigen Peer-Verfahren nur noch ein Datenvolumen von 13 Vektoren besitzt, das heißt 25 % des Datenvolumens von Peer-BAS und 62 % des Datenvolumens von Peer-FUS.

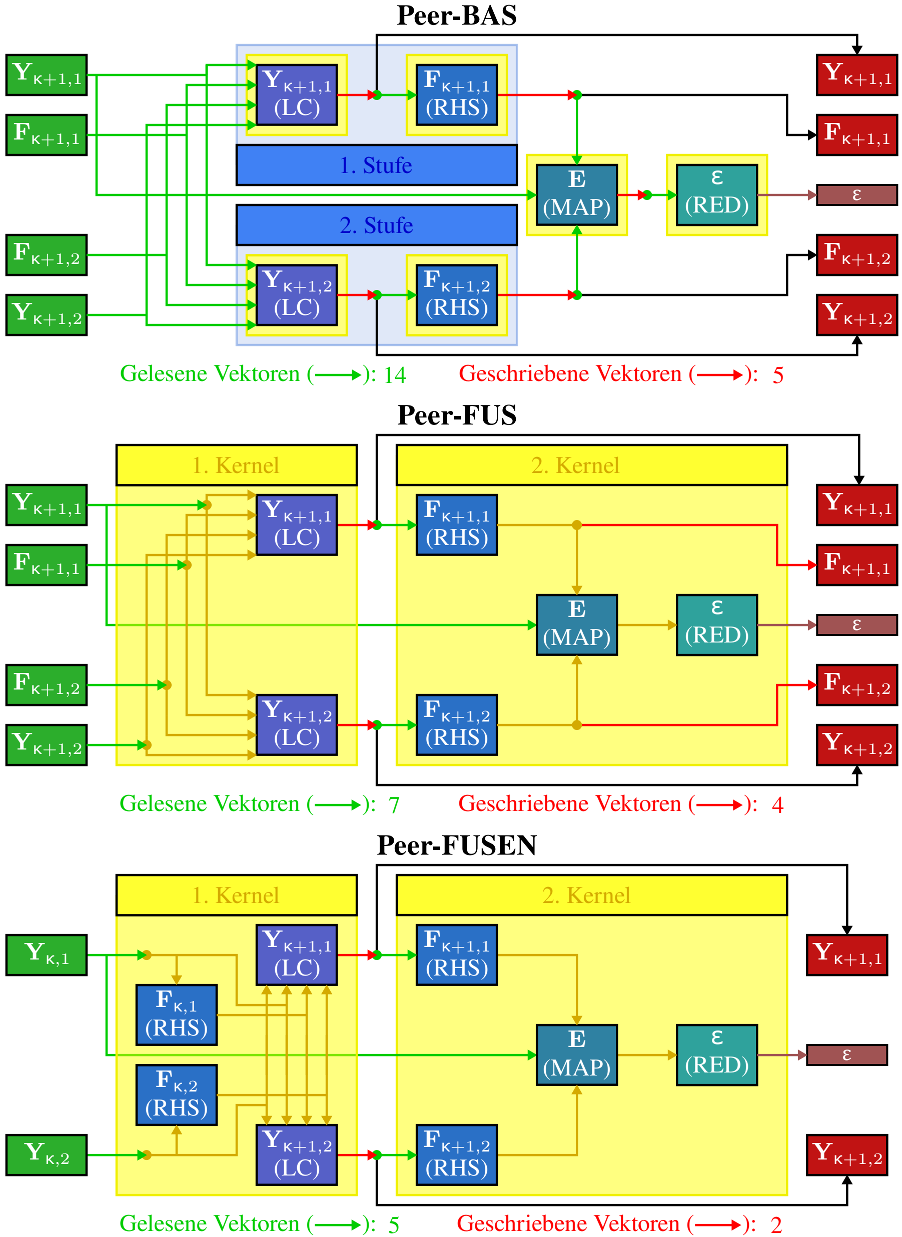


Abbildung 9.9: Datenflussgraphen der Varianten der Peer-Verfahren. Diese Abbildung zeigt beispielhaft die Datenflussgraphen der Varianten für ein vollbesetztes Peer-Verfahren mit zwei Stufen.

Peer-BAS

Kernelstruktur	Berechnungen	Gelesene Daten	Geschriebene Daten
für jede Stufe $i = 1, \ldots, s$			
1. Kernel	$\mathbf{y}_{\kappa+1,i}$	$\mathbf{y}_{\kappa,1:s}, \mathbf{F}_{\kappa,1:s}$	$\mathbf{y}_{\kappa+1,i}$
2. Kernel	$\mathbf{F}_{\kappa+1,i}$	$\mathbf{y}_{\kappa+1,i}$	$\mathbf{F}_{\kappa+1,i}$
3. Kernel	$\mathbf{E}$	$\mathbf{y}_{\kappa,s}, \mathbf{F}_{\kappa+1,1:s}$	$\mathbf{E}$
4. Kernel	$\epsilon_{\kappa+1}$	$\mathbf{E}$	ϵ

Peer-FUS

Kernelstruktur	Berechnungen	Gelesene Daten	Geschriebene Daten
1. Kernel	$\mathbf{y}_{\kappa+1,1:s}$	$\mathbf{y}_{\kappa,1:s}, \mathbf{F}_{\kappa,1:s}$	$\mathbf{y}_{\kappa+1,1:s}$
2. Kernel	$\mathbf{F}_{\kappa+1,1:s}, \mathbf{E}, \epsilon$	$\mathbf{y}_{\kappa,s}, \mathbf{y}_{\kappa+1,1:s}$	$\mathbf{F}_{\kappa+1,1:s}, \epsilon$

Peer-FUSEN

Kernelstruktur	Berechnungen	Gelesene Daten	Geschriebene Daten
1. Kernel	$\mathbf{F}_{\kappa,1:s}, \mathbf{y}_{\kappa+1,1:s}$	$\mathbf{y}_{\kappa,1:s}$	$\mathbf{y}_{\kappa+1,1:s}$
2. Kernel	$\mathbf{F}_{\kappa+1,1:s}, \mathbf{E}, \epsilon$	$\mathbf{y}_{\kappa,s}, \mathbf{y}_{\kappa+1,1:s}$	ϵ

Tabelle 9.11: Kernel der Implementierungsvarianten der Peer-Verfahren.

Variante	#Gelesene Vektoren	#Geschriebene Vektoren	#Summe
Peer-BAS	$2s^2 + 2s + 2$	$2s + 1$	$2s^2 + 4s + 3$
Peer-FUS	$3s + 1$	$2s$	$5s + 1$
Peer-FUSEN	$2s + 1$	s	$3s + 1$

Tabelle 9.12: Zahl der gelesenen und geschriebenen Vektoren der Peer-Verfahren.

9.3 Kernelfusion auf GPUs

9.3.1 Implementierung

9.3.1.1 Grundlegendes

Bei unserer GPU-Implementierung für die per Kernelfusion generierten Varianten übersetzen wir wieder jedes fusionierte abstrakte Kernel zu einem GPU-Kernel und jeden parallelen Block eines abstrakten Kernels zu einer Workgroup. Ein solches GPU-Kernel fokussiert sich auf die Datenwiederverwendung über den Registersatz und besitzt deshalb auch keine Schleifen. Eine solche Datenwiederverwendung über den Registersatz ist besonders vielversprechend auf einer GPU, da ihr Registersatz verglichen mit ihren anderen On-Chip-Speichern sehr groß ist. So ist unter anderem der Registersatz einer GPU meist um ein Vielfaches größer als deren L2-Cache. Zum Beispiel besitzt die GeForce Titan V von NVIDIA einen 20 MiB großen Registersatz und einen 6 MiB großen L2-Cache, während die Radeon RX 480 von AMD einen 8 MiB großen Registersatz und einen 2 MiB großen L2-Cache besitzt.

Eine triviale Datenwiederverwendung über den Registersatz erfordert, dass alle fusionierten Vektorgrundoperationen innerhalb eines Kernels das gleiche Mapping zwischen Vektorkomponenten und Workitems besitzen, da ein jedes Workitem nur auf die Register seiner SIMD-Lane zugreifen kann. Zum Beispiel kann bei diesem Mapping ein Workitem eine von ihm produzierte Komponente einer (RHS | LC | MAP)-Operation zunächst in einem Register abspeichern und dieses Register dann an alle fusionierten, davon abhängigen (LC | MAP | RED)-Operationen weiterleiten. Dadurch kann bei einem solchen Mapping ein jedes Workitem eigenständig die ihm zugewiesenen Komponenten der fusionierten Vektorgrundoperationen in ihrer topologischen Reihenfolge berechnen, ohne sich mit den anderen Workitems seiner Workgroup synchronisieren oder austauschen zu müssen. Wenn dahingegen in einem fusionierten Kernel die Workitems einer Workgroup für unterschiedliche Vektorgrundoperationen unterschiedliche Vektorkomponenten berechnen würden, dann müsste die Workgroup die von ihren Workitems produzierten Komponenten einer (RHS | LC | MAP)-Operation über die Caches, den Scratchpad-Speicher oder SIMD-Shuffle-Instruktionen an fusionierte, davon abhängige (LC | MAP | RED)-Operationen weiterleiten. Zusätzlich müsste eine Workgroup in diesem Fall ihre Workitems untereinander synchronisieren. Hierdurch würde wiederum ein deutlicher Overhead entstehen.

9.3.1.2 Ausnutzung des On-Chip-Speichers

Ein fusioniertes GPU-Kernel kann auf Grund des gleichen Mappings zwischen Vektorkomponenten und Workitems für all seine Vektorgrundoperationen die On-Chip-Speicher einer GPU wie folgt verwenden:

- **Abspeichern von produzierten Vektorkomponenten in den Registern:** Bei der Berechnung einer (RHS | LC | MAP)-Operation speichert ein Workitem beziehungsweise eine SIMD-Lane die von ihm produzierten Vektorkomponenten zunächst in seinen Registern ab.
  - **Transfer von produzierten Vektorkomponenten über die Register:** Das Workitem gibt die Register mit den produzierten Vektorkomponenten an die davon

abhängigen und fusionierten (LC | MAP | RED)-Operationen weiter. Dadurch erfolgt der Datentransfer für fusionierte (RHS | LC | MAP) $\rightarrow$ (LC | MAP | RED)-Abhängigkeiten über den Registersatz der GPU.

– **Kein unnötiges Zurückschreiben von Registern in den DRAM:** Das Workitem schreibt die Register mit den produzierten Vektorkomponenten der (RHS | LC | MAP)-Operation nur in einen permanenten Vektor und damit in den DRAM zurück, falls das Ergebnis dieser (RHS | LC | MAP)-Operation noch von einem folgendem Kernel gelesen wird.

- **Wiederverwendung von Registern, um ein redundantes Laden aus dem DRAM zu vermeiden:** Teilen sich mehrere fusionierte (RHS | LC | MAP | RED)-Operationen einen permanenten Argumentvektor, der nicht vom Kernel selbst produziert wird, sondern zu Beginn des Kernels in dem DRAM abgespeichert ist, so lädt ein Workitem zunächst seine zugewiesenen Komponenten von dem Argumentvektor aus dem DRAM in seine Register. Anschließend gibt es diese Register an alle fusionierten (RHS | LC | MAP | RED)-Operationen weiter. Auf diese Weise kann ein fusioniertes Kernel über die Wiederverwendung von Registern ein redundantes Laden aus dem DRAM vermeiden. Da ein Workitem für die Auswertung einer RHS-Operation je nach Problem aber nicht zwingend nur auf die zugewiesenen, sondern potentiell auf alle Komponenten des Argumentvektors zugreifen muss, kann ein Kernel bei RHS-Operationen nur einen Teil der benötigten Daten über ein Register wiederverwenden.

- **Datenwiederverwendung über die Caches, um ein redundantes Laden aus dem DRAM zu vermeiden:** Teilt sich eine RHS-Operation mit einer fusionierten (LC | MAP | RED)-Operation einen permanenten Argumentvektor aus dem DRAM, dann findet im Kernel, falls die RHS-Funktion lokal auf den Argumentvektor zugreift, eine Datenwiederverwendung über die Caches der GPU statt. Denn in diesem Fall greift ein Workitem für die Auswertung seiner Komponenten der RHS-Operation auf diejenigen Vektorkomponenten zu, die benachbarte Workitems und Workgroups für die Auswertung ihrer Komponenten der (LC | MAP | RED)-Operation benötigen. Somit weisen diese Zugriffe eine räumliche und zeitliche Lokalität auf und können von den Caches der GPU abgefangen werden.

9.3.1.3 Weitere Überlegungen zum Mapping zwischen Vektorkomponenten und Workitems und zu der Dimensionalität eines fusionierten Kernels

Wie wir bereits zuvor feststellten, so müssen alle Vektorgrundoperationen in einem fusionierten Kernel das gleiche Mapping zwischen Vektorkomponenten und Workitems verwenden, damit ein fusioniertes Kernel Daten über den Registersatz der GPU wiederverwenden kann. Ansonsten können wir aber in einem fusionierten Kernel das Mapping zwischen Vektorkomponenten und Workitems und die Dimensionalität des Kernels frei wählen.

Für die Wahl des Mappings und die Dimensionalität des GPU-Kernels sollten wir aber zusätzlich Folgendes bedenken:

- Bei (LC | MAP | RED)-Operationen sollte das Mapping zusätzlich ein perfektes Coalescing ermöglichen. Zudem ist es leicht vorteilhaft, wenn das Mapping bei (LC | MAP | RED)-Operationen die Verwendung von 16-Byte-Gather- und Scatter-Instruktionen ermöglicht, selbst wenn diese bei perfektem Coalescing kaum effizienter sind als Gather-

und Scatter-Instruktionen mit einer geringeren Breite. Sowohl das perfekte Coalescing als auch die Verwendung von 16-Byte-Gather- und Scatter-Instruktionen können wir zum Beispiel bei doppelter Gleitkommagenauigkeit über ein 2:1 Mapping erzielen, welches einem Workitem mit einem globalen Index (i) die beiden Vektorkomponenten $(2 \cdot i)$ und $(2 \cdot i + 1)$ zur Berechnung zuweist.

- Bei (LC | MAP | RED)-Operationen hat die Dimensionalität des Kernels keine Auswirkungen auf die Performance, sofern sie das perfekte Coalescing und die Verwendung von 16-Byte Gather- und Scatter-Instruktionen nicht verhindert.
- Bei RHS-Operationen kann ein Kernel durch ein geeignetes Mapping eine effizientere Vektorisierung verwenden. Zusätzlich kann das Kernel durch ein geeignetes Mapping breitere Speicherzugriffsinstruktionen verwenden, welche bei RHS-Funktionen mit einem nicht perfekten Coalescing deutlich effizienter sind. Zudem kann das Kernel durch ein geeignetes Mapping redundante Berechnung zwischen Systemkomponenten einsparen.
- Bei RHS-Operationen sollte die Dimensionalität des Kernels auf Grund der Speicherzugriffslokalität passend zur räumlichen Struktur des Anfangswertproblems gewählt werden, da dies den Workingset eines Kernels deutlich verkleinert.

Aus diesen Gründen entschieden wir uns für Folgendes:

- Besteht ein fusioniertes Kernel nur aus (LC | MAP | RED)-Operationen, so verwenden wir ein 1D-Kernel und ein 4:1 Mapping zwischen Vektorkomponenten und Workitems für die einfache Gleitkommagenauigkeit sowie ein 2:1 Mapping zwischen Vektorkomponenten und Workitems für die doppelte Gleitkommagenauigkeit.
- Beinhaltet ein fusioniertes Kernel mindestens eine RHS-Operation, so wählen wir für das Kernel diejenige Dimensionalität und dasjenige Mapping zwischen Vektorkomponenten und Workitems, die beide für die RHS-Operation am vorteilhaftesten ist. Beides behalten wir dann auch für alle anderen fusionierten (LC | MAP | RED)-Operationen im Kernel bei. So verwenden wir zum Beispiel für unser BRUSS2D-Testproblem, falls ein fusioniertes Kernel mindestens eine RHS-Operation durchführen sollte, bei einfacher Gleitkommagenauigkeit wieder ein 2D-Kernel mit einem 4:1 Mapping und bei doppelter Gleitkommagenauigkeit ein 2D-Kernel mit einem 2:1 Mapping.

9.3.1.4 Codegenerator für fusionierte GPU-Kernel

Da bei den per Kernelfusion generierten Implementierungsvarianten von ODE-Verfahren sehr viele unterschiedliche fusionierte Kernel entstehen, ist ein händisches Generieren der fusionierten GPU-Kernel sehr aufwändig. Deshalb implementierten wir einen Codegenerator, der für die per Kernelfusion generierten Implementierungsvarianten automatisch die fusionierten GPU-Kernel generiert. Dieser Codegenerator erzeugt dabei nicht direkt einen Maschinencode für die jeweilige Architektur der Ziel-GPU, sondern OpenCL-C oder CUDA-C, welches wir dann an den GPU-Compiler der entsprechenden GPU-API zum Kompilieren weiterleiten. Eine beispielhafte Pseudocode-Abstraktion des generierten Codes für ein fusioniertes GPU-Kernel wird in Abbildung 9.10 gezeigt.

Für die Generierung des Codes eines GPU-Kernels fügt unser Codegenerator diejenigen Vektorgrundoperationen, die in dem Datenflussgraphen zu einem Kernel fusioniert worden sind, gemäß ihrer topologischen Sortierung in den Quelltext des Kernels ein. Zudem realisiert der Codegenerator die Datenwiederverwendung über den Registersatz in den generierten Kernels darüber, dass ein GPU-Compiler automatische Variablen immer in den Registern der GPU alloziert. Im Detail implementiert der Codegenerator die Speicheroptimierungen in einem fusionierten Kernel wie folgt:

- Der Codegenerator setzt den Datentransfer entlang fusionierter (RHS | LC | MAP) $\rightarrow$ (LC | MAP | RED)-Abhängigkeiten über den Registersatz dadurch um, indem er im generierten Code die (RHS | LC | MAP)-Operation am Anfang der Abhängigkeit ihr Ergebnis in eine automatische Variable abspeichern lässt, und diese automatische Variable dann an die fusionierten (LC | MAP | RED)-Operation am Ende der Abhängigkeit weiterleitet.
- Der Codegenerator fügt in den Quelltext des Kernels nur dann ein Statement ein, welches das in einer automatischen Variable gespeicherte, vom Kernel produzierte Ergebnis einer (RHS | LC | MAP)-Operation in den entsprechenden permanenten Vektor und damit in den DRAM zurückschreibt, wenn dieses Ergebnis auch von einem späteren Kernel gelesen wird.
- Der Codegenerator verschmilzt in einem fusionierten Kernel die redundanten Ladeoperationen auf einem permanenten Vektor aus dem DRAM, der von mehreren fusionierten (RHS | LC | MAP | RED)-Operationen als Argument gelesen wird, zu einer einzigen Ladeoperation, indem er ein Workitem mehrmals benötigte Komponenten des permanenten Vektors zunächst in eine automatische Variable laden lässt, und diese automatische Variable dann an alle abhängigen Vektorgrundoperationen weiterleitet. Somit sind unsere fusionierten Kernels für diese Art der Registerwiederverwendung nicht darauf angewiesen, dass der GPU-Compiler diese redundanten Ladeoperationen automatisch als gemeinsame Teilausdrücke erkennt und eliminiert.
- Zudem versieht unser Codegenerator sämtliche Zeiger auf die permanenten Vektoren, die wir dem Kernel als Argumente übergeben, mit dem Schlüsselwort `restrict`. Dies erlaubt es dem GPU-Compiler ein potentielles Aliasing zwischen den Kernelargumenten auszuschließen, und dadurch die Speicherzugriffe stärker zu optimieren (siehe zum Beispiel [T9] oder [T25]).

Typisches Kernel mit fusionierter RHS → LC-Abhängigkeit aus einem RK-Verfahren

```
in: double2* y_k, F_1, Y_2;
out: double2* F_2, Y_2;

parallel for each workgroup wg
   parallel for each workitem wi in wg
      double F_1_reg, F_2_reg, y_k_reg, y_kn_reg;
      int i_x   = global_index_x(wi);
      int i_y   = global_index_y(wi);
      int i     = i_y * size_x + i_x;

      y_k_reg =  y_k[i];
      F_1_reg =  F_1[i];

      F_2_reg = RHS_2D(Y_2, i_x, i_y);
      Y_3_reg = y_k_reg + a_31 * F_1_reg + a_32 * F_2_reg;

      F_2[i] = F_2_reg;
      y_kn[i] = Y_3_reg;
```

Kernel mit zwei fusionierten unabhängigen Stufen aus einem RK-Verfahren ohne Zurückschreiben des Ergebnisses der RHS-Operationen.

```
in: double2* y_k, Y_2, Y_3;
out: double2* Y_4, Y_5;

parallel for each workgroup wg
   parallel for each workitem wi in wg
      double F_1_reg, F_2_reg, y_k_reg, y_kn_reg;
      int i_x   = global_index_x(wi);
      int i_y   = global_index_y(wi);
      int i     = i_y * size_x + i_x;

      y_k_reg =  y_k[i];

      F_2_reg = RHS_2D(Y_2, i_x, i_y);
      F_3_reg = RHS_2D(Y_3, i_x, i_y);
      Y_4_reg = y_k_reg + a_42 * F_2_reg + a_43 * F_3_reg;
      Y_5_reg = y_k_reg + a_52 * F_2_reg + a_53 * F_3_reg;

      Y_4[i] = Y_4_reg;
      Y_5[i] = Y_5_reg;
```

Abbildung 9.10: Schematisierte Beispiele für den generierter GPU-Code für die fusionierten Varianten. In beiden Beispiel verwenden wir ein 2D-Kernel mit einem 2:1 Mapping zwischen Vektorkomponenten und Workitems.

9.3.2 Experimente

9.3.2.1 Aufbau der Experimente

Als Nächstes werden wir die mit Kernelfusion erzeugten Implementierungsvarianten experimentell auf GPUs untersuchen. Diese Experimente bauten wir wie folgt auf: **Betrachtete ODE-Verfahren:** In den Messungen betrachten wir das Verner-Verfahren, das DOPRI-Verfahren, ein PIRK-Verfahren und ein Peer-Verfahren. Als Testfall für die Klasse der PIRK-Verfahren verwenden wir das Lobatto III C(8)-Verfahren mit $s = 5$ Stufen und einer Ordnung von $p = 9$ als Basisverfahren. Dadurch muss das PIRK-Verfahren $m = 7$ Korrektorschritte durchführen. Für die Klasse der Peer-Verfahren betrachten wir in unseren Experimenten ein Peer-Verfahren mit $s = 4$ Stufen. Alle Messungen und Profiling-Ergebnisse normalisieren wir für jedes dieser Verfahren jeweils auf einen einzigen Zeitschritt.

Betrachtete Implementierungsvarianten: Als Implementierungsvarianten für die betrachteten Verfahren verwenden wir diejenigen Varianten, die wir im Abschnitt 9.2.8 ableiteten, das heißt die unfusionierten Varianten der Basisimplementierung (BAS-Varianten), die Varianten nur mit Fusion (FUS-Varianten) und die Varianten mit Fusion und Enabling-Transformationen (FUSEN-Varianten). Dabei dienen die unfusionierten Varianten der Basisimplementierung als Referenz für den Speedup. Wir erzeugten all diese Implementierungsvarianten mit unserem Framework, welches wir im Kapitel 11 im Detail vorstellen werden.

Testproblem und Problemgröße: In diesem Kapitel verwenden wir für alle Messreihen das Testproblem BRUSS2D mit einer Problemgröße von $4\,224 \times 4\,096$ Gitterzellen für Messungen mit der einfachen Gleitkommagenauigkeit und mit einer Problemgröße von $4\,224 \times 2\,048$ Gitterzellen für Messungen mit doppelter Gleitkommagenauigkeit, so dass die Kernels für beide Genauigkeiten das gleiche Datenvolumen verarbeiten. Eine Ausnahme hiervon ist diejenige Messreihe, welche die Skalierung der Laufzeit in Abhängigkeit von der Problemgröße untersucht, für welche wir die Problemgröße schrittweise erhöhen. **Verwendete GPUs und GPGPU-APIs:** Um möglichst viele GPU-Architekturen abzudecken, führen wir unsere Experimente auf drei GPUs mit einer jeweils unterschiedlichen Architektur durch:

- **GeForce GTX 980 Ti (*Maxwell*):** Laufzeitmessungen mit einfacher Gleitkommagenauigkeit, Profiling, Skalierungsuntersuchungen und Vergleich mit ODEINT
- **GeForce Titan Black (*Kepler*):** Laufzeitmessungen mit einfacher und doppelter Gleitkommagenauigkeit
- **Firepro W8100 (*Hawaii*):** Laufzeitmessungen mit einfacher und doppelter Gleitkommagenauigkeit sowie Skalierungsuntersuchungen

Als GPU-API verwenden wir für Maxwell und Kepler die CUDA-Driver-API und für Hawaii die OpenCL-API.

9.3.2.2 Messungen

Insgesamt führten wir folgende Messreihen durch:

- **Laufzeitmessungen (Tabelle 9.13):** Die Messung der Laufzeit unserer Implementierungsvarianten des Verner-Verfahrens, des PIRK-Verfahrens und des Peer-Verfahrens auf Maxwell, Kepler und Hawaii mit einfacher und doppelter Gleitkommagenauigkeit.

- **Speedup (Tabelle 9.14):** Der aus den soeben durchgeführten Laufzeitmessungen berechnete Speedup mit den BAS-Varianten als Referenz.

- **Vergleich mit ODEINT (Tabelle 9.15):** Vergleich der Laufzeit unserer Implementierungsvarianten von DOPRI mit der Implementierungsvariante aus der ODEINT-Bibliothek (DOPRI-ODEINT) auf Maxwell. Da bei DOPRI-ODEINT die Kosten eines Zeitschritts davon abhängen, ob dieser Zeitschritt von der Schrittweitensteuerung akzeptiert wird, betrachten wir hier nur die Kosten für einen akzeptierten Zeitschritt.

- **Profiling (Tabelle 9.16):** Ein detailliertes Profiling der Implementierungsvarianten des Verner-Verfahrens, des PIRK-Verfahrens und des Peer-Verfahrens auf der Maxwell mit einfacher Gleitkommagenauigkeit. Im Speziellen ermittelten wir durch das Profiling die *Auslastung der DRAM Bandbreite* (U_{DRAM}), das *DRAM-Volumen* V_{DRAM}, die *Cache-Effizienz* E_{cache}, die *Auslastung der IPC* U_{IPC} und die *Anzahl der herausgegebenen Instruktionen* (INS). Diese Metriken werden von uns im Abschnitt 13.4 detailliert beschrieben.

- **Skalierung mit der Systemgröße (Abbildung 9.11):** Die Skalierung der inversen normierten Laufzeit mit der Systemgröße für unsere Implementierungsvarianten des Verner-Verfahrens auf Maxwell unter der Verwendung der einfachen Gleitkommagenauigkeit. Für diese Messungen wählten wir eine initiale quadratische Gittergröße von 64×64 Gitterzellen und inkrementierten die Kantenlänge des Gitters um 64 bis zu einer Größe von $4\,096 \times 4\,096$ Gitterzellen. Aus der gemessenen Laufzeit berechneten wir den Durchsatz der GPU an Systemkomponenten pro Zeiteinheit und trugen diesen in das Diagramm ein.

- **Skalierung mit der Anzahl an benutzten GPU-Kerne (Abbildung 9.12):** Die Skalierung der inversen Laufzeit mit der Anzahl an benutzten GPU-Kernen für die Implementierungsvarianten des Verner-Verfahrens, des PIRK-Verfahrens und des Peer-Verfahrens auf Maxwell und Hawaii mit der einfachen Gleitkommagenauigkeit. Für diese Skalierungsuntersuchung verwendeten wir die im Abschnitt 13.3.1 beschriebene Technik, um GPU-Kerne für ein Kernel zu sperren.

Variante	Maxwell	Kepler		Hawaii	
	T_{SP} in [ms]	T_{SP} in [ms]	T_{DP} in [ms]	T_{SP} in [ms]	T_{DP} in [ms]
Verner-BAS	34.9	41.2	41.1	40.9	40.1
Verner-FUS	24.6	27.5	28.9	30.6	29.4
Verner-FUSEN	18.8	21.2	22.1	23.6	24.7
PIRK-BAS	171.2	202.3	202.5	199.9	196.6
PIRK-FUS	48.9	58.1	86.3	75.5	78.1
Peer-BAS	24.1	28.6	28.5	29.1	27.4
Peer-FUS	10.7	12.7	13.7	15.6	14.3
Peer-FUSEN	7.26	9.56	12.9	11.3	11.1

Tabelle 9.13: Laufzeitmessungen für die Varianten mit Kernelfusion auf GPUs. Diese Tabelle zeigt die Laufzeit T für die Varianten des Verner-Verfahrens, des PIRK-Verfahrens und des Peer-Verfahrens auf Maxwell, Kepler und Hawaii mit einfacher Genauigkeit (SP) und doppelter Genauigkeit (DP).

Variante	Maxwell	Kepler		Hawaii	
	S_{SP}	S_{SP}	S_{DP}	S_{SP}	S_{DP}
Verner-BAS	1.00	1.00	1.00	1.00	1.00
Verner-FUS	1.42	1.50	1.42	1.34	1.36
Verner-FUSEN	1.86	1.94	1.86	1.73	1.62
PIRK-BAS	1.00	1.00	1.00	1.00	1.00
PIRK-FUS	3.51	3.48	2.34	2.65	2.52
Peer-BAS	1.00	1.00	1.00	1.00	1.00
Peer-FUS	2.26	2.25	2.08	1.86	1.92
Peer-FUSEN	3.32	2.99	2.21	2.57	2.47

Tabelle 9.14: Speedup für die Varianten mit Kernelfusion auf GPUs. Die Tabelle zeigt den Speedup S, welcher sich aus den Laufzeitmessungen in Tabelle 9.13 ergibt.

	DOPRI-BAS	DOPRI-FUS	DOPRI-FUSEN	DOPRI-ODEINT
T_{SP} in [ms]	25.8	22.9	16.1	34.7
S_{SP}	1	1.12	1.60	0.74

Tabelle 9.15: Vergleich unserer Implementierungsvarianten mit DOPRI-ODEINT auf der Maxwell. Der Vergleich setzt sich aus den gemessenen Laufzeiten T_{SP} aller Varianten und den Speedup S_{SP} bezüglich unserer DOPRI-BAS-Implementierung zusammen.

Variante	U_{DRAM} in [%]	V_{DRAM} in [GB]	E_{cache} in [%]	U_{IPC} in [%]	INS in [10^6]
Verner-BAS	86	10.1	100	5.0	313
Verner-FUS	83	6.9	98	6.0	267
Verner-FUSEN	82	5.2	99	8.8	299
PIRK-BAS	86	48.5	100	5.0	1 539
PIRK-FUS	79	12.9	96	10.7	956
Peer-BAS	87	7.0	100	4.6	198
Peer-FUS	81	2.9	99	7.4	142
Peer-FUSEN	75	1.8	98	14.8	194

Tabelle 9.16: Profiling der Varianten mit Kernelfusion auf Maxwell. Dieses Profiling umfasst der Implementierungsvarianten des Verner-Verfahrens, des PIRK-Verfahrens und des Peer-Verfahrens mit einfacher Gleitkommagenauigkeit. Dabei zeigt dieses Profiling die Metriken der Auslastung der DRAM Bandbreite (U_{DRAM}), das DRAM-Volumen (V_{DRAM}), die Cache-Effizienz (E_{cache}), die Auslastung der IPC (U_{IPC}) und die Zahl der herausgegebenen Instruktionen (INS). Diese Metriken werden von uns im Abschnitt 13.4 detailliert beschrieben.

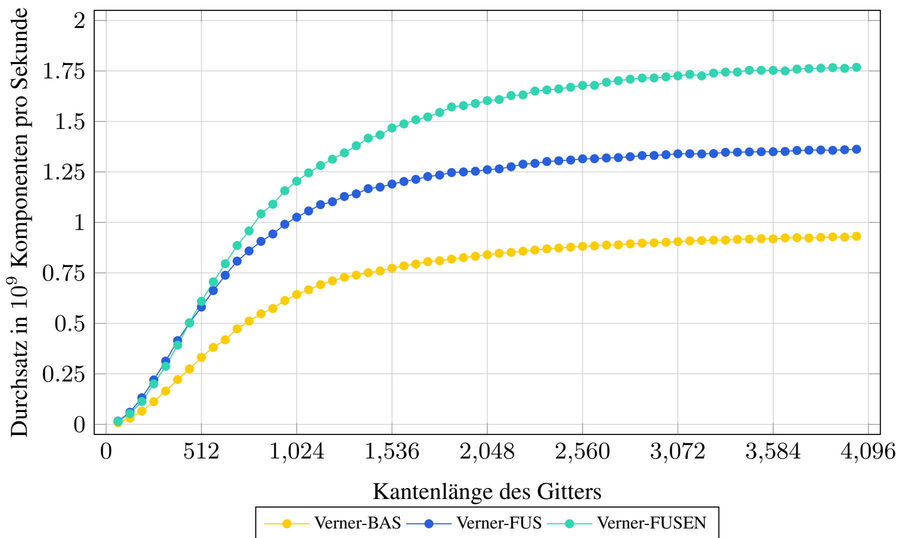


Abbildung 9.11: Skalierung der Laufzeit mit der Gittergröße für die Varianten des Verner-Verfahrens auf Maxwell.

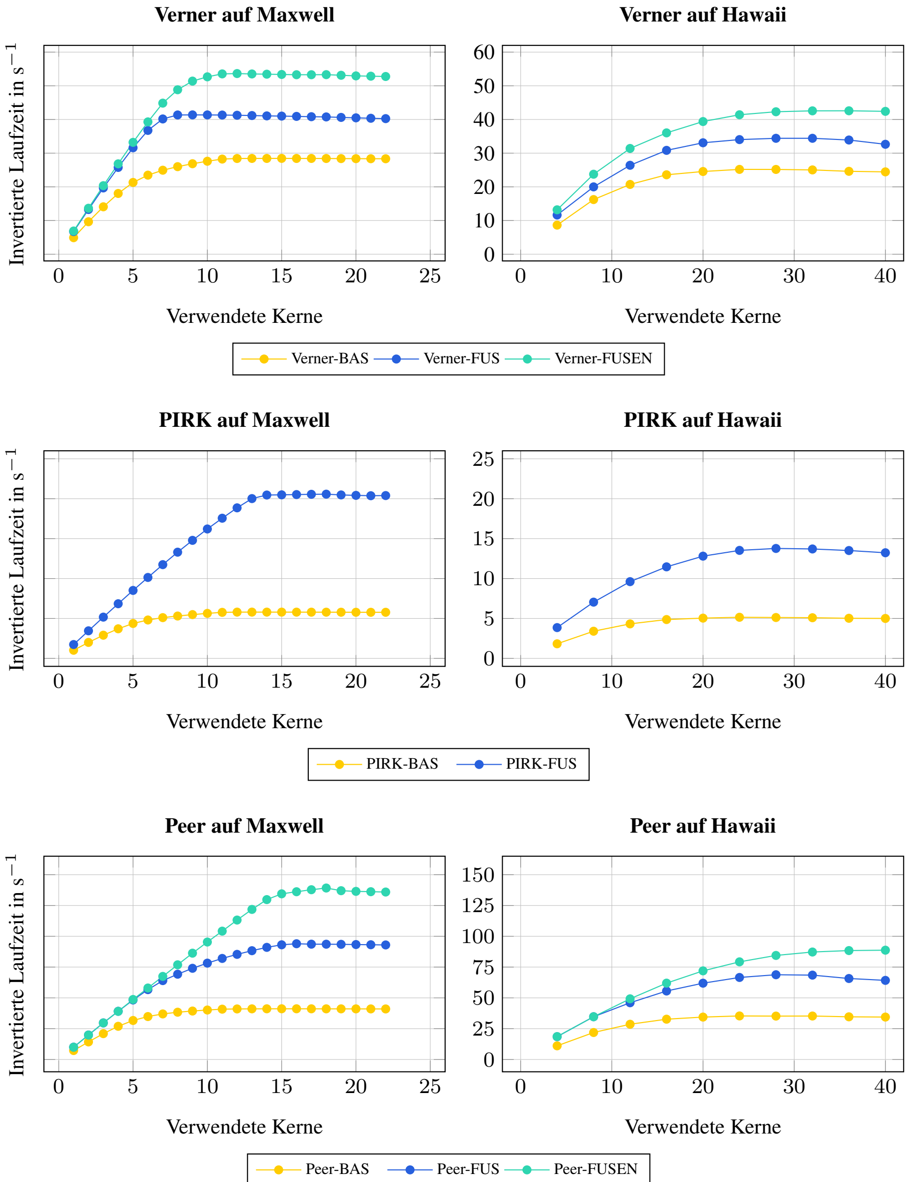


Abbildung 9.12: Skalierung der Varianten mit Kernelfusion mit der Anzahl der verwendeten GPU-Kerne. Für diese Skalierungsuntersuchungen verwendeten wir die einfache Gleitkommagenauigkeit.

9.3.2.3 Diskussion der Messergebnisse

Mit Hilfe dieser Messungen können wir folgende Fragen beantworten:

Wie sehr lohnt sich die Kernelfusion?
(Siehe Tabelle 9.13 und Tabelle 9.14)

Für alle betrachteten Architekturen, betrachteten ODE-Verfahren und beide betrachteten Gleitkommagenauigkeiten sind diejenigen Varianten, die wir nur mit Kernelfusion erzeugten, signifikant schneller als die unfusionierten Varianten der Basisimplementierung. So besitzt Verner-FUS einen Speedup von 1.34 bis 1.5 gegenüber Verner-BAS, PIRK-FUS einen Speedup von 2.34 bis 3.51 gegenüber PIRK-BAS, und Peer-FUS einen Speedup von 1.86 bis 2.26 gegenüber Peer-BAS. Auffällig ist hierbei, dass der Speedup für PIRK-FUS und Peer-FUS auf Kepler mit der doppelten Gleitkommagenauigkeit und auf Hawaii mit beiden Gleitkommagenauigkeiten deutlich geringer ausfällt als auf Maxwell und auf Kepler mit der einfachen Gleitkommagenauigkeit.

Wie sehr lohnen sich die zusätzlichen Enabling-Transformationen?
(Siehe Tabelle 9.13 und Tabelle 9.14)

Die Varianten mit Kernelfusion und Enabling-Transformationen können für alle betrachteten Architekturen, betrachteten ODE-Verfahren und beide betrachteten Gleitkommagenauigkeiten den Speedup gegenüber den Varianten nur mit Kernelfusion, aber ohne Enabling-Transformationen noch einmal signifikant vergrößern: So erzielt Verner-FUSEN einen Speedup von 1.62 bis 1.94 gegenüber der Basisimplementierung, während Verner-FUS nur einen Speedup von 1.34 bis 1.50 erzielt. Ebenso erzielt Peer-FUSEN einen Speedup von 2.21 bis 3.32 gegenüber der Basisimplementierung, während Peer-FUS nur einen Speedup von 1.86 bis 2.26 erzielt. Zudem ist wieder auffällig, dass der Speedup für Peer-FUSEN auf Kepler mit der doppelten Gleitkommagenauigkeit und auf Hawaii mit beiden Gleitkommagenauigkeiten deutlich geringer ausfällt als der Speedup auf Maxwell und Kepler mit der einfachen Gleitkommagenauigkeit.

Wie erfolgreich können die optimierten Varianten das DRAM-Volumen reduzieren?
(Siehe Tabelle 9.16)

Das Profiling auf Maxwell zeigt, dass für alle untersuchten ODE-Verfahren und Architekturen bei den Varianten mit Kernelfusion die GPU signifikant weniger Daten mit dem DRAM austauscht als bei den unfusionierten Varianten der Basisimplementierung. So reduziert Verner-FUS im Vergleich zu Verner-BAS das DRAM-Volumen von 10.1 GB auf 6.9 GB, PIRK-FUS im Vergleich zu PIRK-BAS von 48.5 GB auf 12.9 GB und Peer-FUS im Vergleich zu Peer-BAS von 7.0 GB auf 2.9 GB. Durch die Enabling-Transformationen lässt sich das DRAM-Volumen in Verner-FUSEN und Peer-FUSEN weiter auf 5.2 GB beziehungsweise auf 1.8 GB reduzieren.

Wie effizient kann das Caching das DRAM-Volumen reduzieren?
(Siehe Tabelle 9.16 und Abbildung 9.12)

Das Profiling auf Maxwell zeigt, dass auf dieser GPU für alle Varianten das DRAM-Volumen immer fast gleich groß wie das Datenvolumen ist. Dadurch ergibt sich eine Cache-Effizienz von 97.6 % bis 100.3 % für alle Implementierungsvarianten. Folglich kann das Caching für die FUS-Varianten und für die FUSEN-Varianten weiterhin sehr gut das DRAM-Volumen beinahe auf das Datenvolumen reduzieren, und dies obwohl sowohl die Kernelfusion als auch die Enabling-Transformationen die Workingsets der Kernels vergrößern. Alternativ können wir dies auch aus

der Untersuchung der Skalierbarkeit mit der Kernzahl auf Maxwell und Hawaii folgern. Denn dort erkennen wir, dass bei den FUS-Varianten und bei den FUSEN-Varianten die Performance bei vielen aktiven Kernen nicht für jeden zusätzlichen aktiven Kern abfällt, sondern nahezu konstant bleibt. Einen solchen Abfall würden wir aber erwarten, falls der durch die zusätzlichen aktiven Kerne vergrößerte Workingset nicht mehr in den L2-Cache der GPU passen würde.

Wodurch sind die Varianten gebunden?
(Siehe Tabelle 9.16 und Tabelle 9.14)

Da im Profiling auf Maxwell alle Varianten eine hohe DRAM-Auslastung von 75 % bis 87 % und eine niedrige IPC-Auslastung von 5 % bis 15 % erzielen, können wir folgern, dass auf Maxwell nicht nur die Varianten der Basisimplementierung, sondern auch die Varianten nur mit Kernelfusion sowie die Varianten mit Kernelfusion und Enabling-Transformationen weiterhin sehr stark durch die DRAM-Bandbreite gebunden sind, und die Rechenwerke der GPU nur schlecht auslasten. Da auf Kepler der Speedup bei allen Varianten mit der einfachen Gleitkommagenauigkeit ungefähr direkt proportional zum Datenvolumen ist, können wir dort das Gleiche folgern. Jedoch, da auf Maxwell der Speedup bei PIRK-FUS, Peer-FUS und Peer-FUSEN mit doppelter Gleitkommagenauigkeit und auf Hawaii der Speedup von PIRK-FUS, Peer-FUS und Peer-FUSEN mit beiden Gleitkommagenauigkeiten deutlich geringer ausfällt, als die Reduktion im Datenvolumen es erwarten lässt, und dies nicht auf einen zu großen Workingset für den zu kleinen L2-Cache zurückzuführen ist (siehe vorherige Frage), deutet dies auf einen anderen Flaschenhals, wahrscheinlich die L2-Cache-Bandbreite, hin.

Wie gut kann unser Performance-Modell die Laufzeit vorhersagen?
(Siehe Tabelle 9.16)

Das Profiling zeigt, dass auf Maxwell die beiden Grundannahmen für unser Performance-Modell gut erfüllt sind: Die Performance ist durch die DRAM-Bandbreite gebunden und das Cache-Effizienz beträgt beinahe 100 %. Dadurch ist die Laufzeit in etwa direkt proportional zu dem Datenvolumen der Implementierungsvariante. Es gibt auf Maxwell allerdings eine Unsicherheit, da, obwohl die Implementierungsvarianten stets durch die DRAM-Bandbreite limitiert sind, die Auslastung der DRAM-Bandbreite für die Implementierungsvarianten zwischen 75 % und 86 % schwankt. So würden wir gemäß unserem Performance-Modell bei Verner-FUSEN einen Speedup von 1.97 gegenüber Verner-BAS erwarten, während der tatsächlich gemessene Speedup 1.86 beträgt, bei PIRK-FUS würden wir einen Speedup von 3.91 gegenüber PIRK-BAS erwarten, während der tatsächlich gemessene Speedup 3.51 beträgt, und bei Peer-FUSEN würden wir einen Speedup von 3.92 erwarten, während der tatsächlich gemessene Speedup 3.32 beträgt. Ebenso sind auf Kepler nur alle Varianten mit der einfachen Gleitkommagenauigkeit durch die DRAM-Bandbreite gebunden, weshalb dort der gemessene Speedup ebenfalls in etwa dem erwarteten Speedup entspricht. Dahingegen sind auf Kepler die Varianten mit der doppelten Gleitkommagenauigkeit und auf Hawaii die Varianten mit beiden Gleitkommagenauigkeiten teils durch die L2-Cache-Bandbreite gebunden. In diesem Fall kann unser Performance-Modell die Laufzeit nicht mehr zufriedenstellend vorhersagen.

Wie verändert der Kernelfusion die Anzahl an ausgeführten Instruktionen?
(Siehe Tabelle 9.16)

Bei einer Implementierungsvariante mit Kernelfusion müssen die Kernels für einen Zeitschritt weniger Speicherzugriffsinstruktionen ausführen als bei einer Variante der Basisimplementierung. Zudem kann der Compiler bei den Varianten mit Kernelfusion in den fusionierten Kernels gemeinsame Teilausdrücke bei der Index- und Adressarithmetik entfernen. Deshalb ist davon

auszugehen, dass die Kernelfusion die Gesamtzahl an ausgeführten Instruktionen leicht reduzieren kann. Die Messungen bestätigen dies, da auf Maxwell die Anzahl an ausgeführten Instruktionen von $31 \cdot 10^7$ bei Verner-BAS auf $27 \cdot 10^7$ bei Verner-FUS, von $154 \cdot 10^7$ bei PIRK-BAS auf $94 \cdot 10^7$ bei PIRK-FUS und von $20 \cdot 10^7$ bei Peer-BAS auf $14 \cdot 10^7$ bei Peer-FUS abfällt.

Wie groß ist der Overhead der Cloning-Transformation?
(Siehe Tabelle 9.16)

Das Profiling zeigt, dass obwohl die Cloning-Transformation die Anzahl an RHS-Operationen bei Verner-FUSEN im Vergleich zu Verner-FUS auf 11 von 8 und bei Peer-FUSEN im Vergleich zu Peer-FUS auf 8 von 4 erhöht, die Anzahl an ausgeführten Instruktionen sich, zumindest bei BRUSS2D als Testproblem, nur von $27 \cdot 10^7$ auf $30 \cdot 10^7$ beim Verner-Verfahren und nur von $14 \cdot 10^7$ auf $19 \cdot 10^7$ beim Peer-Verfahren erhöht. Dadurch verursacht bei beiden Verfahren die Cloning-Transformation nur einen Overhead an der Gesamtzahl an ausgeführten Instruktionen von 11 % beziehungsweise 35 %.

Wie skaliert die Performance mit der Zahl der verwendeten GPU-Kerne?
(Siehe Abbildung 9.12)

Die Skalierungsuntersuchungen zeigen sowohl auf Maxwell als auch auf Hawaii für alle Implementierungsvarianten eine typische Sättigungskurve: Für eine kleinere Anzahl an benutzten GPU-Kernen skaliert die Performance beinahe direkt proportional, bis ein Sättigungswert erreicht ist, und weitere zusätzliche GPU-Kerne die Performance immer weniger verbessern. Diese Sättigung beruht auf Maxwell, wie wir aus unserem Profiling in Tabelle 9.16 folgern können, immer auf einer Limitierung durch die DRAM-Bandbreite. Da auf Hawaii der maximale Speedup nur bei Verner-FUS so groß ist, wie man dies im Falle einer Limitierung durch die DRAM-Bandbreite erwarten würde, und bei Verner-FUSEN, PIRK-FUS, Peer-FUS und Peer-FUSEN deutlich kleiner ausfällt, kann dies auf der Hawaii nur von einem anderen Flaschenhals, wahrscheinlich der L2-Cache-Bandbreite, verursacht werden. Bei den Varianten mit Fusion tritt die Sättigung später als bei den unfusionierten Varianten der Basisimplementierung ein. Ebenso tritt bei den Varianten mit Fusion und Enabling-Transformationen die Sättigung noch einmal später ein als bei den Varianten nur mit Fusion, aber ohne Enabling-Transformationen. Bei wie vielen benutzten GPU-Kernen auf einer bestimmten GPU die Sättigung eintritt, ist von der arithmetischen Intensität der Implementierungsvariante abhängig. So tritt beispielsweise auf Maxwell die Sättigung von Verner-BAS bereits bei circa 10 aktiven GPU-Kernen ein, die Sättigung von Verner-FUS bei circa 7 aktiven GPU-Kernen und die Sättigung bei Verner-FUSEN bei circa 10 aktiven GPU-Kernen. Wenn bei einer bestimmten Anzahl von aktiven GPU-Kernen weder die DRAM-Bandbreite noch die L2-Cache-Bandbreite gesättigt ist, würden wir erwarten, dass die Performance in etwa direkt proportional zu der Anzahl an insgesamt ausgeführten Instruktionen ist. Jedoch zeigen unsere Experimente, dass bei einer solchen Anzahl von aktiven GPU-Kernen die unfusionierten Varianten der Basisimplementierung wie erwartet langsamer sind als die Varianten mit Kernelfusion, jedoch die Varianten mit Kernelfusion aber ohne Enabling-Transformationen in etwa genauso schnell sind wie die Varianten mit Kernelfusion und Enabling-Transformationen.

Wie skaliert die Performance mit der Problemgröße?
(Siehe Abbildung 9.11)

Die Messungen auf Maxwell für das Verner-Verfahren zeigen eine asymptotische Kurve für alle drei Implementierungsvarianten für die Skalierung mit der Systemgröße. Für große Systemgrößen erreicht Maxwell einen Durchsatz von $0.93 \cdot 10^9$ Komponenten pro Sekunde bei

Verner-BAS, $1.36 \cdot 10^9$ Komponenten pro Sekunde bei Verner-FUS und von $1.76 \cdot 10^9$ Komponenten pro Sekunde bei Verner-FUSEN. Des Weiteren benötigen alle Varianten ungefähr eine Gittergröße von $2\,200 \times 2\,200$ Zellen, um auf Maxwell 90 % ihres maximalen Durchsatzes zu erzielen. Da Maxwell nur 45 056 Workitems gleichzeitig ausführen kann, welche wiederum nur 90 112 gleichzeitig berechneten Gitterzellen oder einer Gittergröße von circa 300×300 Zellen entsprichen, wird der geringe Durchsatz für kleine Gittergrößen von 300×300 Zellen bis $2\,200 \times 2\,200$ Zellen sehr wahrscheinlich die GPU-CPU-Synchronisation der Schrittweitensteuerung verursacht.

Wie schneiden unsere Implementierungsvarianten im Vergleich zu ODEINT ab?
(Siehe Tabelle 9.15)

Die Messungen mit DOPRI-5 auf Maxwell zeigen, dass, obwohl DOPRI-ODEINT und DOPRI-BAS beinahe die gleiche Anzahl an permanenten Vektoren aus dem DRAM laden und in den DRAM schreiben, DOPRI-BAS einen Speedup von 1.35 gegenüber DOPRI-ODEINT erzielt. Dies beruht hauptsächlich darauf, dass DOPRI-ODEINT die LC-Operationen über ein Template aus der Thrust-Bibliothek von NVIDIA implementiert, welches die DRAM-Bandbreite der GPU deutlich schlechter ausnutzt als die manuell implementierten LC-Operationen. Durch die Kernelfusion und Enabling-Transformationen erzielt DOPRI-FUSEN einen Speedup von 1.60 gegenüber DOPRI-BAS und einen Speedup von 2.16 gegenüber DOPRI-ODEINT.

9.4 Kernelfusion auf CPUs

9.4.1 Implementierung

9.4.1.1 Grundlegendes

Um unsere per Kernelfusion generierten Implementierungsvarianten auf CPUs zu übertragen, übersetzen wir wieder je ein abstraktes Kernel aus unseren theoretischen Überlegungen in ein paralleles Schleifennest. Dieses Schleifennest besteht wieder aus einer parallelen äußeren Schleife, die über die Blöcke des Kernels iteriert. Dabei wird ein jeder Block jeweils nur von einem CPU-Thread berechnet. Zudem weisen wir jedem dieser parallelen Blöcke w_{block} aufeinanderfolgende Systemkomponenten zu, so dass ein Block i alle fusionierten Vektorgrundoperationen des Kernels für die Komponenten $[w_{\text{block}} \cdot i, w_{\text{block}} \cdot (i+1) - 1]$ auswertet. Dieses Blocking zielt dabei auf die Datenverwendung innerhalb eines Blocks über die Caches der CPU ab.

Da wir die fusionierten Kernels nicht allgemein für eine beliebige CPU-Architektur optimieren können, überlegen wir uns im Folgenden wieder nur, wie wir die fusionierten Kernels für moderne x64-Server-CPUs, wie sie zum Beispiel im Abschnitt 3.14 beschrieben wurden, optimieren können. Dabei berücksichtigen wir insbesondere die Vektorisierung und wie sich mit Non-Temporal-Store-Instruktionen der Fetch-On-Write-Overhead des Last-Level-Caches vermeiden lässt. Allerdings lassen sich analoge Überlegungen auch verwenden, um die fusionierten Kernels für andere CPU-Architekturen zu optimieren.

9.4.1.2 Schleifenstrukturen für CPU-Kernel

Während bei dem Schleifennest eines CPU-Kernels die äußerste Schleife immer über die parallelen Blöcke des Kernels iteriert, so kann man sich für die innere Struktur dieses Schleifennests prinzipiell viele Varianten überlegen, welche die Cache-Hierarchie und das Pre-Fetching von modernen CPUs auf eine bestimmte Art ausnutzen. Wir implementierten für diese innere Schleifenstruktur zwei solche Varianten, für welche wir einen Beispielcode in Abbildung 9.13 zeigen:

- **Schleifenstruktur mit einfachem Blocking:** In der *Schleifenstruktur mit einfachem Blocking* beinhaltet das Kernel für jede fusionierte Vektorgrundoperation eine innere Schleife, die die Komponenten des entsprechenden Blocks auswertet. Falls das Anfangswertproblem eine Eval-Range-Implementierung der RHS-Funktion bereitstellt, ersetzen wir diese innere Schleife für jede RHS-Operation durch einen Aufruf dieser Eval-Range-Funktion.
- **Schleifenstruktur mit doppeltem Blocking:** In der *Schleifenstruktur mit doppeltem Blocking* ist jeder Block des Kernels noch einmal in *Subblöcke* unterteilt. Folglich beinhaltet die parallele äußere über die Blöcke iterierende Schleife zunächst eine mittlere Schleife, die über die Subblöcke des aktuellen Blocks iteriert. Diese mittlere Schleife beinhaltet nun für jede fusionierte Vektorgrundoperation eine innere Schleife, die diese Vektorgrundoperation für die Komponenten des jeweiligen Subblocks auswertet. Dabei ersetzen wir bei einer RHS-Operation wieder die innere Schleife durch einen Aufruf der Eval-Range-Funktion, falls diese für das Problem vorhanden ist.

Schleifenstruktur mit einfachem Blocking

```
in: double* y_k, F_1, F_2;
out: double* Y_3, Y_4;

code:
parallel for each block b
   for each component i in b
      Y_3[i] <- y_k[i] + a_31 * F_1[i] + a_32 * F_2[i];
   for each component i in b
      Y_4[i] <- y_k[i] + a_41 * F_1[i] + a_42 * F_2[i];
```

Schleifenstruktur mit doppeltem Blocking

```
in: double* y_k, F_1, F_2;
out: double* Y_3, Y_4;

code:
parallel for each block b
   for each subblock s in b
      for each component i in s
         Y_3[i] <- y_k[i] + a_31 * F_1[i] + a_32 * F_2[i];
      for each component i in s
         Y_4[i] <- y_k[i] + a_41 * F_1[i] + a_42 * F_2[i];
```

Abbildung 9.13: Implementierte Schleifenstrukturen für ein fusioniertes CPU-Kernel. Hierbei symbolisiert der Operator <− Non-Temporal-Store-Instruktionen, durch welche Vektorkomponenten direkt in den DRAM geschrieben werden.

Bei der Schleifenstruktur mit einfachem Blocking besitzen große Blockgrößen einen niedrigeren Overhead und ermöglichen ein effizienteres Pre-Fetching. Jedoch besitzen solche großen Blockgrößen auch ein größeres Workingset, welches unter Umständen zu groß für die niedrigen Cache-Stufen der CPU ist. Die Schleifenstruktur mit doppeltem Blocking vermeidet dieses Problem, da wir für die Blockgröße der äußeren Schleife, um den Overhead zu reduzieren und ein effizientes Pre-Fetching zu ermöglichen, einen hohen Wert wählen können, während wir für die Größe der Subblöcke der inneren Schleife einen kleinen Wert wählen können, so dass der Workingset eines Subblocks in den L1-Cache eines CPU-Kerns passt.

Falls wir jedoch eine große Systemdimension annehmen, dann wird die Schleifenstruktur mit einfachem Blocking ungefähr ein genauso großes DRAM-Volumen benötigen, wie die Schleifenstruktur mit doppeltem Blocking oder eine beliebige andere innere Schleifenstruktur. Wenn wir jedoch noch zusätzlich annehmen, dass ein fusioniertes Kernel weiterhin durch die DRAM-Bandbreite limitiert wird, werden sämtliche Schleifenstrukturen für ein fusioniertes Kernel ungefähr dieselbe Laufzeit benötigen. Da wir aber von der Schleifenstruktur mit doppeltem Blocking erwarten, dass sie die niedrigeren Cache-Ebenen der CPU effizienter ausnutzt, wird sie wahrscheinlich die DRAM-Bandbreite mit weniger verwendeten CPU-Kernen saturieren als

die Schleifenstruktur mit einfachem Blocking. Dadurch ist die Schleifenstruktur mit doppeltem Blocking, falls ein Benutzer nur ausreichend viele Kerne verwendet, um die DRAM-Bandbreite zu saturieren, auch energieeffizienter und ein Benutzer kann die nicht benötigten Kerne für andere Berechnungen verwenden.

9.4.1.3 Ausnutzung der Caches der CPU für die Wiederverwendung von Daten

Als Nächstes werden wir vorstellen, wie die Blöcke eines fusionierten Kernels auf einer CPU deren Caches zur Reduktion des DRAM-Volumens ausnutzen können. Hierfür nehmen wir wieder eine große Systemgröße an, so dass ein Vektor deutlich größer als die Caches der CPU ist, wodurch nur sehr wenig Datenwiederverwendung zwischen zwei aufeinanderfolgenden Kernels möglich ist. In diesem Fall können die Blöcke eines fusionierten Kernels die Caches der CPU wie folgt für die Reduktion des DRAM-Volumens ausnutzen:

- **Datentransfer entlang fusionierter (RHS | LC | MAP) $\rightarrow$ (LC | MAP | RED)-Abhängigkeiten via den Caches der CPU (siehe Abbildung 9.14):** Für den Datentransfer entlang fusionierter (RHS | LC | MAP) $\rightarrow$ (LC | MAP | RED)-Abhängigkeiten über die Caches der CPU verwenden wir eine Technik, welche als *Array-Contraction* (siehe zum Beispiel [K35]) bekannt ist. Hierfür schreibt der Thread das Ergebnis der (RHS | LC | MAP)-Operation am Anfang der Abhängigkeit für die Komponenten seines aktuellen Blocks, ohne dabei Non-Temporal-Store-Instruktionen zu verwenden, in ein kleines *temporäres Array*, welches privat für den jeweiligen Thread ist. Der Thread übergibt danach bei der Berechnung seines aktuellen Blocks das temporäre Array an sämtliche abhängigen Vektorgrundoperationen, welche dann während ihrer Auswertung die zuvor berechneten Komponenten aus dem temporären Array lesen. Nachdem der Thread schließlich seinen aktuellen Block vervollständigt hat, verwendet er das temporäre Array wieder, um die berechneten Komponenten der (RHS | LC | MAP)-Operation für seinen nächsten Block zwischenzuspeichern und an alle abhängigen Vektorgrundoperationen weiterzuleiten. Unter der Annahme, dass der Workingset eines Blocks hinreichend klein ist, wird dieses temporäre Array nicht während eines Blocks oder zwischen zwei Blöcken, sondern erst nach dem Ende des Kernels aus den Caches der CPU verdrängt. Dadurch schreibt das Kernel für diesen Datentransfer keine Komponenten der (RHS | MAP | LC)-Operation am Anfang der Abhängigkeit in den DRAM zurück. Da dieser Datentransfer per Array-Contraction über die Caches der CPU nur ohne Non-Temporal-Store-Instruktionen möglich ist, darf der Thread nur mit regulären Store-Instruktionen in das temporäre Array hinschreiben. Somit, falls eine RHS-Operation über eine Eval-Range-Funktion in das temporäre Array schreiben soll, muss hierfür zwingend eine Implementierung für die Eval-Range-Funktion ohne Non-Temporal-Store-Instruktionen verwendet werden.
- **Zurückschreiben der Ergebnisse von (RHS | LC | MAP)-Operationen über Non-Temporal-Store-Instruktionen in den DRAM (siehe Abbildung 9.14):** In demjenigen Fall, dass ein späteres Kernel das Ergebnis einer (RHS | LC | MAP)-Operation benötigt, muss dasjenige Kernel, welches diese (RHS | LC | MAP)-Operation berechnet, deren Ergebnis per Non-Temporal-Store-Instruktionen in den permanenten Vektor und damit in den DRAM zurückschreiben, um so den Fetch-On-Write-Overhead des Last-Level-Caches der CPU zu vermeiden.

- **Wiederverwendung eines permanenten Vektors aus dem DRAM, der von mehreren verschmolzenen Vektorgrundoperationen als Argument gelesen wird:** Hier lädt ein Block die benötigten Komponenten des Argumentvektors zunächst bei der Auswertung der ersten darauf zugreifenden Vektorgrundoperation aus dem DRAM in die Caches der CPU. Da der Block für die Auswertung seiner zweiten Vektorgrundoperation dieselben Komponenten laden muss, gehen diese Zugriffe bei dieser Auswertung nicht mehr auf den DRAM, sondern über, sondern werden von den Caches der CPU abgefangen.

Somit muss eine (RHS | LC | MAP)-Operation beim Abspeichern ihres Ergebnisses folgende Fälle berücksichtigen:

- Falls ihr Ergebnis nur von den fusionierten Vektorgrundoperationen in ihrem Kernel, aber von keiner Vektorgrundoperation in einem späteren Kernel gelesen wird, muss sie ihr Ergebnis per reguläre Store-Instruktionen in ein temporäres Array schreiben.
- Falls ihr Ergebnis von keiner fusionierten Vektorgrundoperation in ihrem Kernel, aber von einer Vektorgrundoperation in einem späteren Kernel gelesen wird, muss sie ihr Ergebnis per Non-Temporal-Store-Instruktionen in einen permanenten Vektor schreiben.
- Falls ihr Ergebnis sowohl von einer fusionierten Vektorgrundoperation in ihrem Kernel als auch von einer Vektorgrundoperation in einem späteren Kernel gelesen wird, muss sie ihr Ergebnis sowohl per reguläre Store-Instruktionen in ein temporäres Array als auch per Non-Temporal-Store-Instruktionen in einen permanenten Vektor schreiben.

Bei (LC | MAP)-Operationen und bei einer RHS-Operation mit der Eval-Comp-Implementierung kann unser Codegenerator für die fusionierten CPU-Kernel (siehe Abschnitt 9.4.1.4) diese Fallbehandlung eigenständig durchführen. Dahingegen muss bei einer Eval-Range-Implementierung für eine RHS-Operation diese Fallbehandlung von dieser Eval-Range-Implementierung selbst übernommen werden.

Letztlich gilt wieder anzumerken, dass bei den Varianten mit Kernelfusion die Non-Temporal-Store-Instruktionen für das Schreiben in permanente Vektoren wieder nur bei großen Problemgrößen lohnenswert sind, während dafür bei kleinere Problemgrößen reguläre Store-Instruktionen besser geeignet sind. Solche Varianten mit Kernelfusion und regulären Store-Instruktionen für das Schreiben in permanente Vektoren lassen sich jedoch auch mit unserem Ansatz erzeugen.

9.4.1.4 Codegenerator für fusionierte CPU-Kernels

Da bei den per Kernelfusion generierten Implementierungsvarianten von ODE-Verfahren für CPUs ebenso sehr viele unterschiedliche fusionierte CPU-Kernel entstehen, integrierten wir einen Codegenerator für diese fusionierten CPU-Kernels in unser Framework, welcher anhand der fusionierten Kernels in einem Datenflussgraphen deren C-Quelltext erzeugt. Dabei beherrscht der Codegenerator sämtliche Optimierungen, die wir in diesem Kapitel vorstellten. Nach der Generierung des Quelltexts durch den Codegenerator kann unser Framework zu seiner Laufzeit diesen Quelltext mit einem C-Compiler zunächst zu einer dynamischen Bibliothek kompilieren, diese dann laden und anschließend das Kernel darüber ausführen.

Wiederverwendung von F_1 ohne Zurückschreiben von F_1 sowie Zurückschreiben von Y_2 ohne Wiederverwendung von Y_2

```
in: double* y_k;
temp: double F_1_Temp[threadcount][blocksize];
out: double* Y_2;

code:
int my_id = get_thread_id();
parallel for each block b
   for each component i in b
      F_1_Temp[my_id][i_rel(b,i)] =  RHS_Eval_Comp(y_k, i);
   for each component i in b
      Y_2[i] <- y_k[i] + a_21 * F_1_Temp[my_id][i_rel(b,i)];
```

Wiederverwendung von F_1 mit Zurückschreiben von F_1 sowie Zurückschreiben von Y_2 ohne Wiederverwendung von Y_2

```
in: double* y_k;
temp: double F_1_Temp[threadcount][blocksize];
out: double* F_1, Y_2;

code:
int my_id = get_thread_id();
parallel for each block b
   for each component i in b
      double F_1_Reg =  RHS_Eval_Comp(y_k, i);
      F_1_Temp[my_id][i_rel(b,i)] = F_1_Reg;
      F_1[i] <- F_1_Reg;
   for each component i in b
      Y_2[i] <- y_k[i] + a_21 * F_1_Temp[my_id][i_rel(b,i)];
```

Abbildung 9.14: Pseudocode für die Datenwiederverwendung per Array-Contraction und für das Zurückschreiben per Non-Temporal-Stores. Da ein Thread das Array `F_1_Temp` zwischen den Blöcken wiederverwendet, wird es während eines Kernels nicht aus den Caches verdrängt, wodurch es auch nicht von der CPU in den DRAM zurückgeschrieben wird. Somit lässt sich dadurch, bei einem CPU-Kernel das Zurückschreiben von produzierten Vektorkomponenten in den DRAM vermeiden. Zusätzlich symbolisiert der Operator `<-` wieder Non-Temporal-Store-Instruktionen, durch welche Vektorkomponenten direkt in den DRAM geschrieben werden.

So kann der Codegenerator einen C-Quelltext mit der Schleifenstruktur des einfachen Blockings und mit der Schleifenstruktur des doppelten Blockings erzeugen. Dabei sind die verwendeten Block-Größen der beiden Schleifenstrukturen Compilezeitkonstanten, so dass der verwendete C-Compiler diese Schleifenstrukturen bei der späteren Kompilierung abrollen oder vektorisieren kann. Zusätzlich beherrscht der Codegenerator die Datenwiederverwendung über temporäre Arrays beziehungsweise per Array-Contraction, so dass das Kernel keine Daten unnötig in

den DRAM zurückschreibt. Ebenso setzt er das Schreiben des Kernels in permanente Vektoren per Non-Temporal-Store-Instruktionen um, indem er je nach dem unterstützten Befehlssatz der Zielarchitektur AVX2-Intrinsics oder AVX-512-Intrinsics in den Quelltext einfügt. Allerdings kann der Codegenerator für kleine Problemgrößen auch so konfiguriert werden, dass er das Schreiben des Kernels in permanente Vektoren über reguläre Store-Instruktionen realisiert, für welche er je nach dem unterstützten Befehlssatz wiederum AVX2-Intrinsics oder AVX-512-Intrinsics verwendet. Zusätzlich versieht er sämtliche Zeiger wieder mit dem Schlüsselwort `restrict`, damit der verwendete C-Compiler bei der späteren Kompilierung des Kernels das Aliasing zwischen den Argumenten des Kernels ausschließen und somit die Speicherzugriffe auf die permanenten Vektoren besser optimieren kann.

9.4.2 Experimente

9.4.2.1 Aufbau der Experimente

Als Nächstes werden wir die mit Kernelfusion erzeugten Implementierungsvarianten experimentell auf CPUs untersuchen. Damit unsere experimentellen Untersuchungen der Kernelfusion auf GPUs und auf CPUs möglichst vergleichbar sind, verwenden wir einen Aufbau für unsere Experimente, der sehr ähnlich zu dem Aufbau zur experimentellen Untersuchung der Kernelfusion auf GPUs ist:

Betrachtete ODE-Verfahren: Als ODE-Verfahren betrachten wir das Verner-Verfahren und ein PIRK-Verfahren. Für dieses PIRK-Verfahren verwenden wir wieder das Lobatto III C(8)-Verfahren mit $s = 5$ Stufen und einer Ordnung von $p = 8$ als Basisverfahren. Dadurch müssen wir für das PIRK-Verfahren $m = 7$ Korrektorschritte durchführen. Alle Messungen und Profiling-Ergebnisse normalisierten wir für jedes dieser Verfahren jeweils auf einen einzigen Zeitschritt.

Betrachtete Implementierungsvarianten: Als Implementierungsvarianten für die betrachteten Verfahren verwenden wir diejenigen Varianten, die wir im Abschnitt 9.2.8 ableiteten, das heißt die unfusionierten Varianten der Basisimplementierung (BAS-Varianten), die Varianten nur mit Fusion (FUS-Varianten) und die Varianten mit Fusion und Enabling-Transformationen (FUSEN-Varianten). Wir realisierten über unseren Codegenerator sowohl die FUS- als auch die FUSEN-Varianten jeweils mit beiden Schleifenstrukturen, also mit der Schleifenstruktur mit einfachem Blocking und der Schleifenstruktur mit doppeltem Blocking. Um jede Messungen jeweils mit einer performanten Blockgröße und Subblockgröße durchzuführen, bestimmten wir vor jeder Messung beides über eine vollständige Suche über eine vorgegebene Menge an Blockgrößen und Subblockgrößen. Unsere Kernels verwenden für sämtliche Untersuchungen Non-Temporal-Store-Instruktionen, um in die permanenten Vektoren zu schreiben. Eine Ausnahme hiervon ist diejenige Messreihe, welche die Skalierung mit der Systemgröße unter der Verwendung von regulären Store-Instruktionen untersucht. Weiterhin verwenden wir die unfusionierten Varianten der Basisimplementierung wieder als Referenz für den Speedup. Wir erzeugten all diese Implementierungsvarianten mit unserem Framework, welches wir im Kapitel 11 im Detail vorstellen werden.

Testproblem, Problemgröße und Genauigkeit: In diesem Kapitel verwenden wir für alle Messreihen das Testproblem BRUSS2D und die doppelte Gleitkommagenauigkeit. Wir wählten zudem dieselbe Problemgröße wie bei unseren experimentellen Untersuchungen der Kernelfusion mit doppelter Gleitkommagenauigkeit auf GPUs, und zwar eine Problemgröße von $4\,224 \times 2\,048$ Gitterzellen. Dadurch sind bei einem ODE-Verfahren auch die auf den GPUs und CPUs gemessenen Laufzeiten miteinander vergleichbar. Eine Ausnahme von dieser fest gewählten Problemgröße sind diejenigen Messreihen, welche die Skalierung der Laufzeit in Abhängigkeit von der Problemgröße untersuchen, für welche wir die Gittergröße ausgehend von 128×128 Zellen schrittweise um 128×128 Zellen bis zu einer Gittergröße von $4\,096 \times 4\,096$ erhöhen. **Verwendete CPUs, Compiler und Compileroptionen:** Wir führen unsere Experimente auf zwei CPUs mit unterschiedlichen Architekturen durch:

- **Skylake Core i9-7980XE mit 8 $\times$ 16 GiB PC4-17000 DDR4-Modulen (*Skylake*):** Laufzeitmessungen, Profiling, Untersuchung der Skalierung mit der Systemgröße und die Untersuchung der Skalierung mit der Anzahl an verwendeten CPU-Kernen

- **Ryzen 5950x mit 4 × 16 GiB PC4-23466 DDR4-Modulen (*Ryzen*):** Laufzeitmessungen und Untersuchung der Skalierung mit der Anzahl an verwendeten CPU-Kernen

Durch diese DRAM-Bestückung ergibt sich eine Burst-DRAM-Bandbreite von 68 GB/s auf Skylake und von 46.9 GB/s auf Ryzen. Die weiteren technischen Daten beider CPUs sind im Abschnitt 13.6 zu finden. Da beide CPUs zweifaches SMT besitzen, starten wir, um das SMT in unseren Messungen auszunutzen, stets zwei Threads pro benutzten CPU-Kern.

Da Voruntersuchungen ergaben, dass tendenziell sowohl der beliebte *MSVC-Compiler* von Microsoft [W18] als auch der ebenfalls beliebte quelloffene *GCC-Compiler* [W19] die Schleifenstrukturen in dem Quelltext unserer Kernels überhaupt nicht oder nur sehr ineffizient vektorisieren, verwendeten wir für die Kompilierung unserer Kernels sowohl auf Skylake als auch auf Ryzen den *ICC-Compiler* von Intel [W17] mit der höchsten Optimierungsstufe (`-O3`). Damit dieser ICC-Compiler unsere Kernels für Skylake optimiert und dabei vor allem die SIMD-Instruktionen von AVX-512 in den Maschinencode unserer Kernels einfügt, verwendeten wieder dieselben Compiler-Optionen wie für die Kompilierung der Basisimplementierung (`-march=skylake-avx512` und `-qopt-zmm-usage=high`). Für Ryzen wiesen wir den Compiler an, Maschinencode mit den SIMD-Instruktionen von AVX2 zu generieren (`-march=core-avx2`).

9.4.2.2 Messungen

Insgesamt führten wir folgende Messungen durch:

- **Laufzeitmessungen und Speedup (Tabelle 9.17):** Die Messung der Laufzeit unserer Implementierungsvarianten des Verner-Verfahrens sowie des PIRK-Verfahrens auf Skylake und Ryzen mit der Schleifenstruktur mit einfachem Blocking sowie mit der Schleifenstruktur mit doppeltem Blocking und der aus der Laufzeit berechnete Speedup.
- **Profiling (Tabelle 9.18):** Ein detailliertes Profiling der Implementierungsvarianten des Verner-Verfahrens sowie des PIRK-Verfahrens auf Skylake. Im Speziellen ermittelten wir durch das Profiling die *Auslastung der DRAM-Bandbreite* (U_{DRAM}), das *DRAM-Volumen* (V_{DRAM}), die *Cache-Effizienz* (E_{cache}), die *Ausnutzung der DP-Peak-Performance* ($U_{\text{DP-PEAK}}$) und die *Anzahl an herausgegebenen Instruktionen* (INS). Diese Metriken werden von uns im Abschnitt 13.4 detailliert beschrieben.
- **Skalierung mit der Systemgröße (Abbildung 9.15 und Abbildung 9.16):** Die Skalierung der inversen normierten Laufzeit mit der Systemgröße für unsere Implementierungsvarianten des Verner-Verfahrens und des PIRK-Verfahrens auf der Skylake. Dabei verwenden wir in der Messreihe in Abbildung 9.15 Non-Temporal-Store-Instruktionen, um in die permanenten Vektoren zu schreiben, während wir in der Messreihe in Abbildung 9.16 mit regulären Store-Instruktionen in die permanenten Vektoren schreiben. Für diese Messungen wählten wir eine initiale quadratische Gittergröße von 128 × 128 Gitterzellen und inkrementierten die Kantenlänge des Gitters um 128 bis zu einer Größe von 4 096 × 4 096 Gitterzellen. Aus der gemessenen Laufzeit berechneten wir den Durchsatz der CPU an Systemkomponenten pro Zeiteinheit und trugen diesen in das Diagramm ein.
- **Skalierung mit der Anzahl an benutzten Kerne (Abbildung 9.17):** Die Skalierung der inversen Laufzeit mit der Anzahl an benutzten CPU-Kernen für die Implementierungsvarianten des Verner-Verfahrens und des PIRK-Verfahrens auf Skylake und auf Ryzen.

Zu den Messungen ist Folgendes anzumerken:

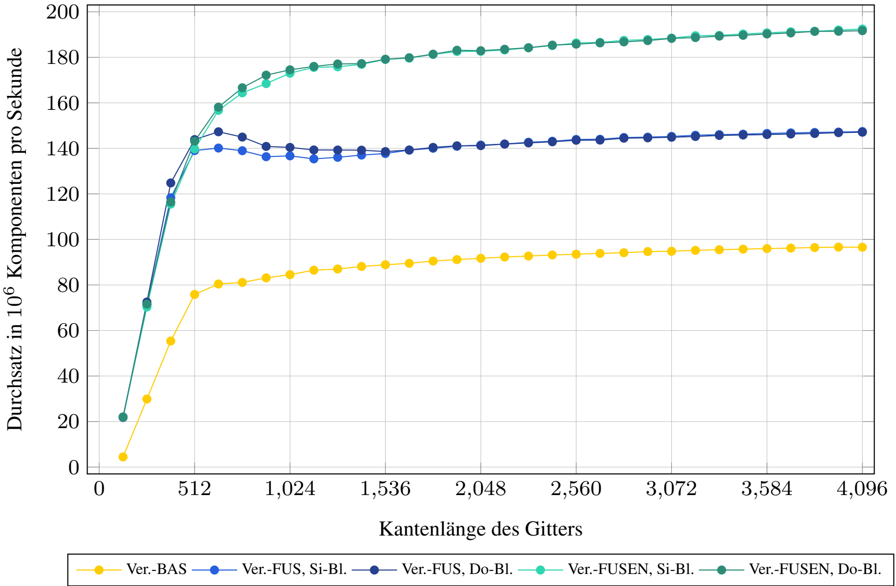
- Wir benutzten für alle Messungen sämtliche Kerne der entsprechenden Test-CPU, selbst wenn dies zu einer Erhöhung der Laufzeit führt. Eine Ausnahme hiervon ist offensichtlich die Untersuchung der Skalierung mit der Anzahl an benutzten Kernen.
- Für die Untersuchung der Skalierung mit der Anzahl an benutzten Kernen starten wir jeweils zwei Threads pro zu benutzenden physikalischen CPU-Kern und weisen die Threads den logischen Kernen des entsprechenden physikalischen CPU-Kerns fest zu.

Variante	Skylake		Ryzen	
	T in [ms]	S	T in [ms]	S
Verner-BAS	182.6	1.00	333.6	1.00
Verner-FUS, Si-Bl.	119.2	1.53	212.9	1.57
Verner-FUS, Do-Bl.	119.7	1.53	202.5	1.65
Verner-FUSEN, Si-Bl.	92.0	1.98	168.5	1.98
Verner-FUSEN, Do-Bl.	92.1	1.98	155.6	2.14
PIRK-BAS	915.9	1.00	1633.7	1.00
PIRK-FUS, Si-Bl.	221.2	4.14	416.4	3.92
PIRK-FUS, Do-Bl.	220.5	4.14	393.7	4.15

Tabelle 9.17: Messung von Laufzeit und Speedup für die Varianten mit Kernelfusion auf CPUs. Die Tabelle zeigt die Laufzeit T und den Speedup s für die Varianten des Verner-Verfahrens und des PIRK-Verfahrens mit den Schleifenstrukturen mit einfachem Blocking (Si-Bl.) und mit doppeltem Blocking (Do-Bl.) auf Skylake und Ryzen mit doppelter Genauigkeit.

Variante	U_{DRAM} in [%]	V_{DRAM} in [GB]	E_{cache} in [%]	U_{DP} in [%]	U_{IPC} in [%]	INS in $[10^6]$
Verner-BAS	75	10.6	95	0.8	1.0	685
Verner-FUS, Si-Bl	80	7.1	96	1.2	2.1	961
Verner-FUS, Do-Bl	82	7.2	95	1.2	2.5	1 165
Verner-FUSEN, Si-Bl	80	5.4	95	1.9	2.3	825
Verner-FUSEN, Do-Bl	73	5.4	92	1.8	2.7	992
PIRK-BAS	71	51.6	94	0.9	1.0	1 843
PIRK-FUS, Si-Bl	75	12.9	96	3.5	3.5	3 085
PIRK-FUS, Do-Bl	76	13.3	99	3.4	1.9	3 373

Tabelle 9.18: Profiling der Varianten mit Kernelfusion auf dem Skylake. Dieses Profiling umfasst die Implementierungsvarianten des Verner-Verfahrens und des PIRK-Verfahrens und des PIRK-Verfahrens mit doppelter Genauigkeit. Im Speziellen ermittelten wir durch das Profiling die Metriken der Auslastung der DRAM Bandbreite (U_{DRAM}), das DRAM-Volumen (V_{DRAM}), die Cache-Effizienz (E_{cache}), die Ausnutzung der DP-Peak-Performance (U_{DP}) und die Zahl der herausgegebenen Instruktionen (INS). Diese Metriken werden von uns im Abschnitt 13.4 detailliert beschrieben.



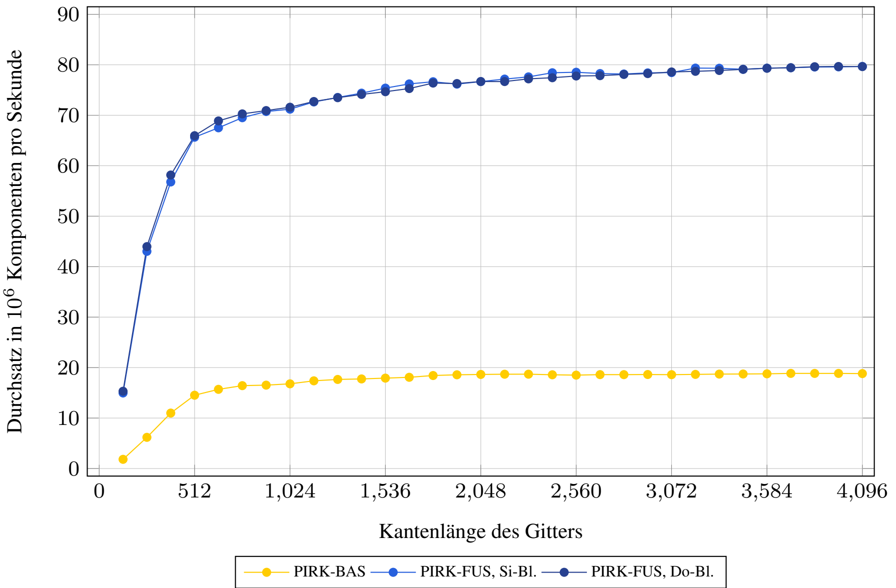
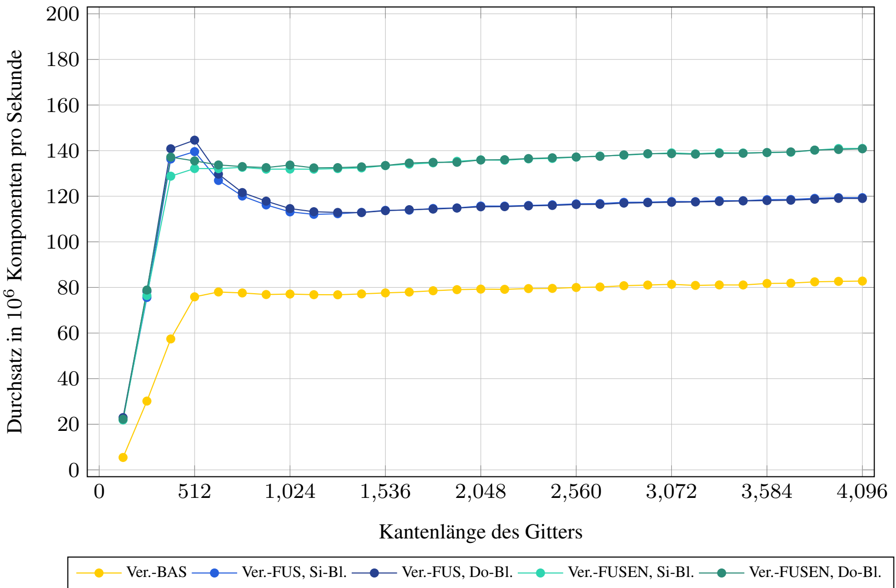


Abbildung 9.15: Skalierung des Durchsatzes mit der Gittergröße für die Varianten des Verner-Verfahrens und des PIRK-Verfahrens mit Non-Temporal-Store-Instruktionen auf Skylake.



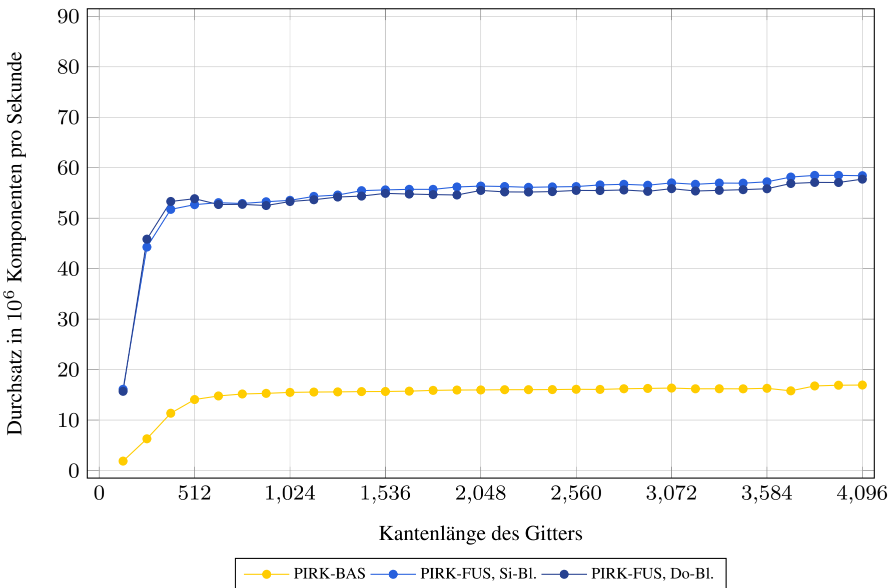


Abbildung 9.16: Skalierung des Durchsatzes mit der Gittergröße für die Varianten des Verner-Verfahrens und des PIRK-Verfahrens mit regulären Store-Instruktionen auf Skylake.

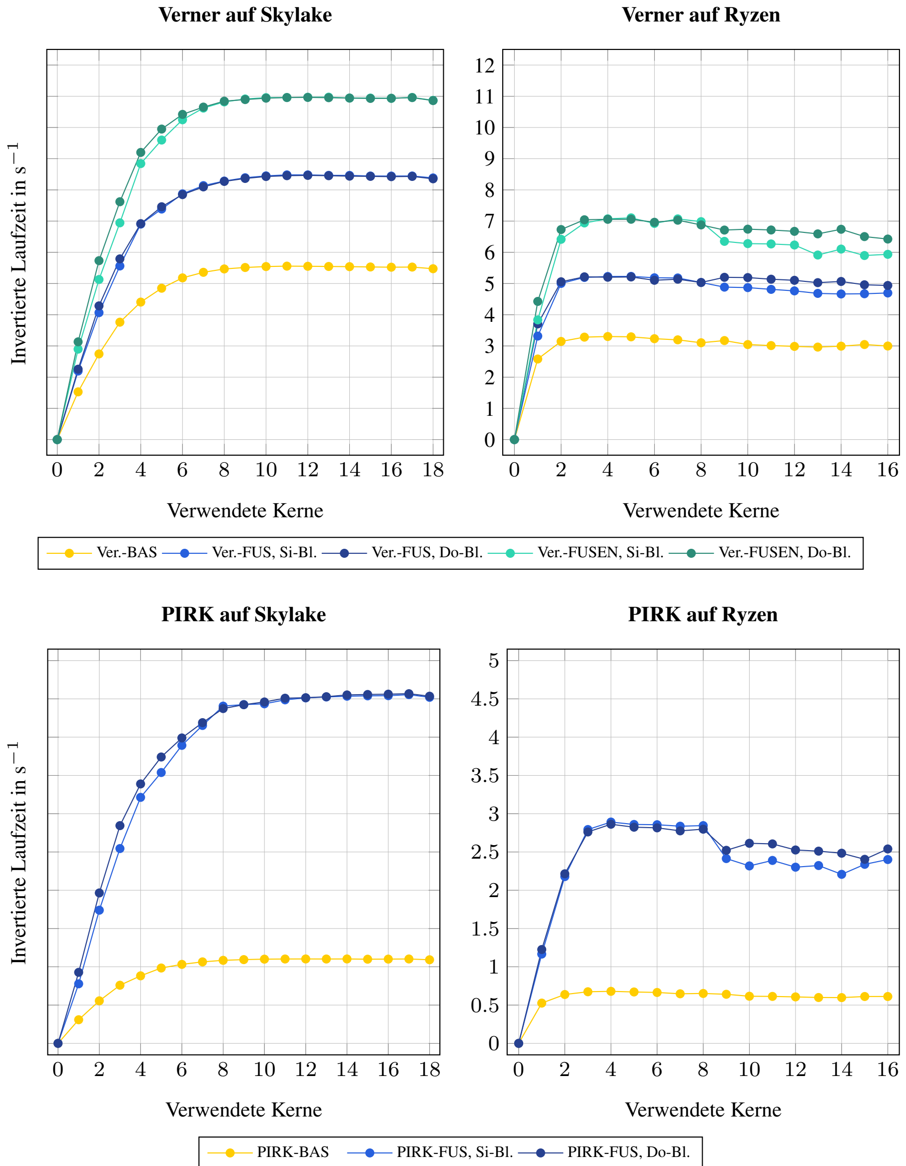


Abbildung 9.17: Skalierung der Varianten mit Kernelfusion mit der Anzahl der verwendeten CPU-Kerne.

9.4.2.3 Diskussion der Messergebnisse

Mit Hilfe dieser Messungen können wir folgende Fragen beantworten:

Wie sehr lohnt sich die Kernelfusion auf CPUs?
(siehe Tabelle 9.17)

Die Kernelfusion kann auf Skylake für das Verner-Verfahren die Laufzeit von 183 ms bei Verner-BAS auf 119 ms bei Verner-FUS und für das PIRK-Verfahren die Laufzeit von 915 ms bei PIRK-BAS auf 221 ms bei PIRK-FUS reduzieren, wodurch sich ein Speedup von 1.53 für das Verner-Verfahren und ein Speedup von 4.14 für das PIRK-Verfahren ergibt. Ebenso kann die Kernelfusion auf Ryzen die Laufzeit für das Verner-Verfahren von 334 ms bei Verner-BAS auf 203 ms bei Verner-FUS und für das PIRK-Verfahren die Laufzeit von 1634 ms bei PIRK-BAS auf 394 ms bei PIRK-FUS senken, womit sich ein Speedup von 1.65 für das Verner-Verfahren und ein Speedup von 3.92 für das PIRK-Verfahren ergibt. Somit kann die Kernelfusion auf beiden betrachteten CPUs und für beide betrachtete Verfahren die Laufzeit deutlich reduzieren.

Wie sehr lohnen sich die zusätzlichen Enabling-Transformationen auf CPUs?
(siehe Tabelle 9.17)

Die Cloning- und die Splitting-Transformation können auf Skylake für das Verner-Verfahren die Laufzeit von 119 ms bei Verner-FUS auf 92 ms bei Verner-FUSEN senken, wodurch sich für Verner-FUSEN ein Speedup von 1.29 bezüglich Verner-FUS ergibt. Ebenso können die Cloning- und die Splitting-Transformation für das Verner-Verfahren auf Ryzen die Laufzeit von 203 ms bei Verner-FUS auf 156 ms bei Verner-FUSEN senken, wodurch sich für Verner-FUSEN ein Speedup von 1.30 bezüglich Verner-FUS ergibt. Somit lohnen sich auch die Enabling-Transformationen, um auf beiden betrachteten CPUs die Laufzeit deutlich zu reduzieren.

Wie erfolgreich können die optimierten Varianten auf CPUs das DRAM-Volumen reduzieren?
(siehe Tabelle 9.18)

Das Profiling zeigt, dass Verner-BAS 10.6 GB an Daten zwischen CPU und DRAM transferiert, während Verner-FUS nur noch 7.1 GB an Daten transferiert. Verner-FUSEN reduziert die Menge an transferierten Daten noch einmal auf 5.4 GB. Somit benötigt Verner-FUSEN als am stärksten optimierte Variante nur noch 51 % des DRAM-Volumens von Verner-BAS. Ebenso transferiert PIRK-BAS 51.6 GB an Daten zwischen CPU und DRAM, während PIRK-FUSEN nur noch 12.9 GB an Daten transferieren muss. Dadurch benötigt PRIK-FUS nur noch 25 % des DRAM-Volumens von PIRK-BAS.

Wie effizient kann das Caching das DRAM-Volumen reduzieren?
(siehe Tabelle 9.18)

Das Profiling auf Skylake zeigt, dass die Cache-Effizienz für sämtliche betrachtete ODE-Verfahren und Implementierungsvarianten zwischen 92 % und 96 % beträgt, das heißt ein perfektes Caching würde das DRAM-Volumen nur um weitere 4 % bis 9 % reduzieren. Da unsere CPU-Kernel jedoch sehr lokal mit einem kleinen Workingset auf den Arbeitsspeicher zugreifen, würden wir jedoch auf CPUs genauso wie auf den GPUs eine Cache-Effizienz von beinahe 100 % erwarten. Mögliche Ursachen dafür, dass die Cache-Effizienz auf CPUs niedriger als erwartet ist, sind Speicherzugriffe des Betriebssystems, welche im Hintergrund der Benchmarks stattfinden, oder ein Overhead durch das Profiling.

Wie skaliert die Performance mit der Anzahl an benutzten CPU-Kernen?
(siehe Abbildung 9.17)

Die Skalierungsuntersuchungen zeigen sowohl auf Skylake als auch auf Ryzen eine typische Sättigungskurve, die darauf hindeutet, dass eine geteilte Ressource, also entweder die Bandbreite des geteilten Last-Level-Caches oder des DRAMs, zum Flaschenhals wird. Auf Skylake können wir aus dem Profiling folgern, dass es sich bei dem Flaschenhals um die DRAM-Bandbreite handeln muss, während es sich bei Ryzen, da er keinen geteilten Last-Level-Cache besitzt, beim Flaschenhals nur um die DRAM-Bandbreite handeln kann. So skalieren auf Skylake die Implementierungsvarianten des Verner-Verfahrens und des PIRK-Verfahrens bis circa 3 oder 4 benutzten CPU-Kernen nahezu linear, und sie müssen nur zwischen 7 und 8 CPU-Kerne verwenden, um 95 % ihrer maximalen Performance zu erreichen. Die Skalierungsuntersuchungen zeigen zudem, dass, sofern die DRAM-Bandbreite kein Flaschenhals ist, bei allen Implementierungsvarianten ein Kern von Ryzen ungefähr 1.5- bis 2.0-mal so schnell wie ein Kern von Skylake ist. Dies ist insoweit bemerkenswert, als ein Kern von Ryzen eine DP-Peak-Performance von 48 GFLOPs besitzt, während ein Kern von Skylake nur eine DP-Peak-Performance von 89 GFLOPs besitzt. Daraus können wir folgern, dass, selbst wenn die DRAM-Bandbreite kein Falschenhals ist, ein Kern von Skylake bei allen Implementierungsvarianten nur einen deutlich kleineren Anteil seiner DP-Peak-Performance ausnutzen kann als ein Kern von Ryzen. Ebenso zeigen die Skalierungsuntersuchungen, dass bei wenigen verwendeten CPU-Kernen, wenn die DRAM-Bandbreite kein Flaschenhals ist, die FUS-Varianten deutlich schneller als die BAS-Varianten und die FUSEN-Varianten zudem deutlich schneller als die FUS-Varianten sind.

Wodurch sind die Varianten gebunden?
(siehe Tabelle 9.18 und Abbildung 9.17)

Das Profiling zeigt, dass auf Skylake sämtliche Varianten zwischen 71 % und 82 % der Burst-DRAM-Bandbreite und nur zwischen 1 % und 3 % der Peak-IPC auslasten. Dies deutet darauf hin, dass sämtliche Varianten weiterhin stark durch die DRAM-Bandbreite gebunden sind. Diese Limitierung durch die DRAM-Bandbreite äußert sich sowohl auf Skylake als auch auf Ryzen dadurch, dass bei sämtlichen Implementierungsvarianten nur ein einziger oder wenige CPU-Kerne benötigt werden, um die maximale Performance zu erzielen. Dementsprechend können auf Skylake auch sämtliche die Varianten nur zwischen 0.8 % und 3.5 % der Peak-DP-Performance ausnutzen. Allerdings lässt sich durch die Kernelfusion und die Enabling-Transformationen die Ausnutzung der Peak-DP-Performance deutlich von 0.8 % bei Verner-BAS auf 1.8 % bei Verner-FUSEN und von 0.9 % bei PIRK-BAS auf 3.5 % bei PIRK-FUS erhöhen.

Wie gut kann unser Performance-Modell die Laufzeit vorhersagen?
(siehe Tabelle 9.17, Tabelle 9.18 und Abbildung 9.17)

Das Profiling und die Skalierungsuntersuchungen zeigen, dass die Grundannahmen für unser Performance-Modell auch auf CPUs gut erfüllt sind: Das Caching funktioniert nahezu perfekt und sämtliche Varianten sind stark durch die DRAM-Bandbreite gebunden. Dadurch entspricht der Speedup zwischen zwei Varianten wiederum in etwa dem Verhältnis zwischen den Datenvolumen dieser beiden Varianten. So würden wir bei Verner-FUS, Verner-FUSEN und PIRK-FUS einen Speedup von 1.49, 1.97, beziehungsweise 3.91 bezüglich der Basisimplementierungen Verner-BAS und PIRK-BAS erwarten, während der tatsächlich gemessene Speedup auf Skylake 1.53, 1.98, beziehungsweise 4.14 und auf Ryzen 1.65, 2.14 beziehungsweise 4.15 beträgt.

Wie sehr lohnt sich die Schleifenstruktur mit doppeltem Blocking verglichen mit der Schleifenstruktur mit einfachem Blocking?
(siehe Tabelle 9.17)

Die Messungen zeigen, dass für alle betrachteten ODE-Verfahren und Implementierungsvarianten, sowohl auf Skylake als auch auf Ryzen bei wenigen aktiven CPU-Kernen, die Schleifenstruktur mit doppeltem Blocking um 5 % bis 10 % schneller ist, als die Schleifenstruktur mit einfachem Blocking. Dies ist vermutlich darauf zurückzuführen, dass, da bei wenigen aktiven CPU-Kernen die FUS- und FUSEN-Varianten noch nicht so stark durch die DRAM-Bandbreite gebunden sind, die Datenwiederverwendung über die niederen Cache-Ebenen eine größere Rolle spielt. Auf Skylake gilt zudem, dass je mehr CPU-Kerne eine Implementierungsvariante verwendet, umso geringer wird auch der Laufzeitunterschied zwischen beiden Schleifenstrukturen, so dass sich bei mehr als 8 verwendeten CPU-Kernen die Laufzeit der beiden Schleifenstrukturen nur noch um weniger als 0.5 % unterscheidet. Dies entspricht auch unseren Erwartungen, da beide Schleifenstrukturen das gleiche Datenvolumen besitzen, und somit, falls das Kernel durch die DRAM-Bandbreite gebunden ist, gleich schnell sein sollten. Auf Ryzen zeigt sich jedoch, dass dort bei vielen aktiven CPU-Kernen sowohl beim Verner-Verfahren als auch beim PIRK-Verfahren die Schleifenstruktur mit doppeltem Blocking weiterhin um 5 % bis 10 % schneller ist als die Schleifenstruktur mit einfachem Blocking. Die Ursache hierfür geht aus unseren Messungen jedoch nicht hervor. Ebenso zeigt das Profiling auf Skylake, dass bei sämtlichen Implementierungsvarianten die Schleifenstruktur mit doppeltem Blocking 10 % bis 20 % mehr Instruktionen ausführt als die Schleifenstruktur mit einfachem Blocking. Da wir für das Profiling eine Blockgröße von 4 096 und eine Subblockgröße von 1 024 verwendeten, sollte der Overhead für das doppelte Blocking vernachlässigbar sein. Deshalb deutet ein großer Unterschied in der Anzahl an ausgeführten Instruktionen darauf hin, dass der ICC-Compiler beide Schleifenstrukturen deutlich unterschiedlich optimiert.

Welchen Einfluss bewirken die Kernelfusion und die Enabling-Transformationen auf die Anzahl an ausgeführten Instruktionen?
(siehe Tabelle 9.18)

Das Profiling zeigt, dass die Kernelfusion auf Skylake beim Verner-Verfahren die Anzahl an ausgeführten Instruktionen um 40 % und beim PIRK-Verfahren sogar um 67 % erhöht. Dieser große Unterschied deutet wieder darauf hin, dass der ICC-Compiler die Kernel der FUS-Varianten verglichen mit den Kernels der BAS-Varianten deutlich unterschiedlich optimiert. Interessanterweise reduzieren die zusätzlichen Enabling-Transformationen bei Verner-FUSEN die Anzahl an ausgeführten Instruktionen verglichen mit Verner-FUS sogar um 15 %, und dies, obwohl Verner-FUSEN auf Grund der Cloning-Transformation 11 RHS-Operationen pro Zeitschritt durchführt, während Verner-FUS nur 8 RHS-Operationen pro Zeitschritt durchführt. Dies ist wiederum ein Hinweis darauf, dass der ICC-Compiler die FUS- und die FUSEN-Varianten deutlich unterschiedlich optimiert.

Wie skaliert die Performance mit der Problemgröße, wenn die Kernel Non-Temporal-Store-Instruktionen verwenden, um in permanente Vektoren zu schreiben?
(siehe Abbildung 9.15)

Die Untersuchungen für die Skalierung des Durchsatzes an Komponenten mit der Systemgröße auf Skylake zeigen für die Implementierungsvarianten des Verner-Verfahrens eine asymptotische Kurve. Für große Systemgrößen erreicht Skylake einen maximalen Durchsatz von $98 \cdot 10^6$ Komponenten pro Sekunde bei Verner-BAS, $1.46 \cdot 10^6$ Komponenten pro Sekunde bei

Verner-FUS und $192 \cdot 10^6$ Komponenten pro Sekunde bei Verner-FUSEN. Um 90 % dieses maximalen Durchsatzes zu erreichen, benötigt Skylake eine Gittergröße von $1\,500 \times 1\,500$ Zellen bei Verner-BAS, eine Gittergröße von 500×500 Zellen bei Verner-FUS und eine Gittergröße von $1\,000 \times 1\,000$ Zellen bei Verner-FUSEN. Dieser geringe Durchsatz für kleine Systemgrößen ist auf den Overhead des Thread-Pools unseres Frameworks (siehe Abschnitt 11.9.3) von circa 0.21 ms für jeden Kernelstart zurückzuführen. Da sich bei unseren Implementierungen des Verner-Verfahrens die LC-Operationen von zwei aufeinanderfolgenden Kernels viele permanente Argumentvektoren teilen, und somit bei kleinen Systemdimensionen trotz der Verwendung von Non-Temporal-Store-Instruktionen auch Datenwiederverwendung zwischen zwei aufeinanderfolgenden Kernels stattfindet, würden wir erwarten, dass der Durchsatz bei kleinen Systemgrößen ein lokales Maximum erreicht, und bei etwas größeren Systemgrößen, für welche die Datenwiederverwendung zwischen aufeinanderfolgenden Kernels abnimmt, ebenfalls abfällt. Dieser Effekt ist, auf Grund des großen Overheads des Thread-Pools für kleine Systemgrößen, jedoch nur bei Verner-FUS zu beobachten, da hier Durchsatz bei einer Gittergröße von circa 600×600 Zellen ein lokales Maximum aufweist. Dieses lokale Maximum tritt insbesondere bei der Schleifenstruktur mit doppeltem Blocking, wahrscheinlich auf Grund ihrer erhöhten Zugriffslokalität, deutlich hervor. Für die Implementierungsvarianten des PIRK-Verfahrens zeigen die Skalierungsuntersuchungen ebenso eine asymptotische Kurve, wobei der maximale Durchsatz bei PIRK-BAS $19 \cdot 10^6$ Komponenten pro Sekunde und bei PIRK-FUS $80 \cdot 10^6$ Komponenten pro Sekunde beträgt, und sowohl PIRK-BAS als auch PIRK-FUS eine Gittergröße von ungefähr $1\,000 \times 1\,000$ Zellen benötigen um 90 % dieses maximalen Durchsatzes zu erreichen. Da wir für diese Messreihe Non-Temporal-Store-Instruktionen verwendeten, um die berechneten Systemzustände in die permanenten Vektoren zurückzuschreiben, existiert bei PIRK-FUS selbst für kleinere Systemdimensionen potentiell nur eine Wiederverwendung vom Systemzustand $\mathbf{y}_\kappa$ zwischen zwei aufeinanderfolgenden Kernels, welcher auch nur circa 9 % zum Datenvolumen eines Kernels beiträgt. Dementsprechend entspricht es den Erwartungen, dass die Skalierungsuntersuchung beim PIRK-Verfahren kein lokales Maximum bei einer kleinen Systemgröße aufzeigen.

Wie skaliert die Performance mit der Systemgröße, wenn die Kernels statt Non-Temporal-Store-Instruktionen reguläre Store-Instruktionen verwenden, um ihr Ergebnis in permanente Vektoren zu schreiben?
(Siehe Abbildung 9.15 und Abbildung 9.16)

Wenn auf Skylake die Kernels für das Schreiben in die permanenten Vektoren reguläre Store-Instruktionen statt Non-Temporal-Store-Instruktionen verwenden, erhöht sich bei den Implementierungsvarianten des Verner-Verfahrens der Durchsatz an Komponenten für Gittergrößen kleiner als ungefähr 500×500 Zellen um circa 15 %, während sich der Durchsatz an Komponenten bei den Implementierungsvarianten des PIRK-Verfahrens für Gittergrößen kleiner als ungefähr 400×400 Zellen um nur ungefähr 5 % erhöht. Diese nur geringe Erhöhung des Durchsatzes bei beiden Verfahren wird dadurch verursacht, dass bei solch kleinen Gittergrößen der Overhead des Threadpools unseres Frameworks von circa 0.21 ms für jeden Kernelstart die Laufzeit dominiert. Bei Gittergrößen größer als ungefähr $1\,000 \times 1\,000$ Zellen benötigen auf Skylake sowohl sämtliche Implementierungsvarianten des Verner-Verfahrens als auch des PIRK-Verfahrens mit regulären Store-Instruktionen eine 10 % bis 30 % größere Laufzeit als die entsprechende Implementierungsvariante mit Non-Temporal-Store-Instruktionen. Bei einer Gittergröße von ungefähr 300×300 bis 500×500 Zellen weist nun der Durchsatz an Komponenten pro Sekunde von Verner-FUS (mit einfachem Blocking), Verner-FUSEN (mit einfachem und doppeltem Blocking) ein deutliches lokales Maximum auf. Ebenso weist der Durchsatz von

PIRK-FUS (mit doppeltem Blocking) ein kleines lokales Maximum bei einer Gittergröße von ungefähr 500×500 Zellen auf. Dahingegen sind für Verner-BAS und PIRK-BAS keine solchen lokalen Maxima erkennbar.

Welche Unterschiede und Gemeinsamkeiten zeigt die experimentelle Untersuchung der Kernelfusion und der Enabling-Transformationen auf GPUs und CPUs?
(siehe Abschnitt 9.3.2.3)

Sowohl auf den untersuchten GPUs als auch auf den untersuchten CPUs sind sämtliche durch die Kernelfusion und die Enabling-Transformationen generierten Implementierungsvarianten stark durch die DRAM-Bandbreite gebunden, und nutzen die Rechenleistung des jeweiligen Prozessors nur schlecht aus. Allerdings können auf allen untersuchten GPUs und CPUs die Caches das DRAM-Volumen sehr effizient reduzieren, weshalb das Profiling auf Skylake und auf Maxwell eine Cache-Effizienz von 94 % bis 100 % ergibt. Aus diesen Gründen ist die Laufzeit einer Implementierungsvariante eines Verfahrens sowohl auf einer GPU als auch auf einer CPU in etwa direkt proportional zum Datenvolumen der Implementierungsvariante, und in etwa indirekt proportional zur maximalen DRAM-Bandbreite des jeweiligen Prozessors. So würden wir beispielhaft erwarten, da das Verhältnis der Burst-DRAM-Bandbreiten von Maxwell und Skylake circa 4.95:1 beträgt (337 GB/s gegenüber 68 GB/s), dass eine Implementierungsvariante auf Maxwell auch circa 4.95-mal so schnell ist wie auf Skylake. Tatsächlich beträgt das Verhältnis zwischen den Laufzeiten auf Maxwell und auf Skylake zum Beispiel bei Verner-FUSEN 1:4.89 (18.8 ms gegenüber 92.0 ms) und bei PIRK-FUS 1:4.51 (48.9 ms gegenüber 220.5 ms).

Da der Flaschenhals der DRAM-Bandbreite auf modernen CPUs deutlich enger als auf modernen GPUs ist (zum Beispiel 0.026 Bytes per SP-FLOP auf Skylake gegenüber 0.059 Bytes per SP-FLOP auf Maxwell), können die Implementierungsvarianten auf CPUs nur einen kleineren Anteil der Peak-Performance ausnutzen als auf GPUs. So beträgt zum Beispiel bei Verner-FUSEN die Auslastung der IPC auf Maxwell 8.8 %, während die Auslastung der IPC auf Skylake nur 2.3 % und die Ausnutzung der Peak-DP-Performance nur 1.9 % beträgt.

Auch zeigen die Experimente, dass die Anzahl an Instruktionen, die ein Prozessor für den Zeitschritt einer Implementierungsvariante ausführen muss, sowohl auf GPUs und CPUs dieselbe Größenordnung aufweist. Allerdings sind unsere Profiling-Untersuchungen in Tabelle 9.16 und in Tabelle 9.18 nicht direkt miteinander vergleichbar, da wir für das Profiling auf Maxwell eine doppelt so große Problemgröße wählten wie für das Profiling auf Skylake, das heißt, um eine Vergleichbarkeit zwischen den Untersuchungen herzustellen, müssen wir die Zahl der von Maxwell ausgeführten Instruktionen halbieren. Zusätzlich müssen wir für einen Vergleich noch berücksichtigen, dass wir für das Profiling auf Maxwell die einfache Gleitkommagenauigkeit verwendeten, für welche die Maxwell eine logische Vektorbreite von 32 Komponenten besitzt, während wir für das Profiling auf Skylake die doppelte Gleitkommagenauigkeit verwendeten, für welche Skylake eine logische Vektorbreite von 8 Komponenten besitzt. Unter der Berücksichtigung der unterschiedlichen Systemgrößen und Vektorbreiten würden wir erwarten, dass das Verhältnis der Anzahl an ausgeführten Instruktionen zwischen Skylake und Maxwell circa 2:1 beträgt. Bei Verner-BAS beträgt das gemessene Verhältnis der Anzahl an ausgeführten Instruktionen zwischen Skylake und Maxwell 2.18:1 ($685 \cdot 10^6$ Instruktionen auf Skylake gegenüber $313 \cdot 10^6$ Instruktionen auf Maxwell), bei Verner-FUSEN 2.75:1 ($825 \cdot 10^6$ Instruktionen auf Skylake gegenüber $299 \cdot 10^6$ Instruktionen auf Maxwell), bei PIRK-BAS 1.19:1 ($1\,843 \cdot 10^6$ Instruktionen auf Skylake gegenüber $1\,539 \cdot 10^6$ Instruktionen auf Maxwell) und bei PIRK-FUSEN 3.22 : 1 ($3\,085 \cdot 10^6$ Instruktionen auf Skylake gegenüber $956 \cdot 10^6$ Instruktionen auf Maxwell).

Die Untersuchungen der Skalierung der Performance mit der Systemgröße zeigen, dass CPUs verglichen mit GPUs nur eine deutlich kleinere Gittergröße benötigen, um 90 % ihres maximalen Durchsatzes zu erreichen. So kann zum Beispiel für Verner-FUSEN Skylake bereits 90 % seines maximalen Durchsatzes ab einer Gittergröße von $1\,000 \times 1\,000$ Zellen erzielen, während Maxwell hierfür mindestens eine Gittergröße von $2\,200 \times 2\,200$ Zellen, das heißt ein 4.8-mal so großes ODE-System, benötigt.

Zusammenfassend können wir somit sagen, dass die experimentellen Untersuchungen der Kernelfusion auf GPUs und CPUs ein sehr ähnliches Gesamtbild zeigen, und sich sowohl die Gemeinsamkeiten als auch die Unterschiede größtenteils über die Eckdaten der Architekturen (DP-Peak-Performance, DRAM-Bandbreite, Vektorbreiten) erklären lassen.

9.5 Fazit

In diesem Kapitel zeigten wir, dass wir das über die Technik der Kernelfusion optimierte Implementierungsvarianten von expliziten ODE-Verfahren generieren können, welche gegenüber der Basisimplementierung aus dem 8. Kapitel ein deutlich verringertes Datenvolumen beziehungsweise eine deutlich vergrößerte Datenwiederverwendung besitzen, wodurch sie auch für dünnbesetzte Anfangswertprobleme mit einer niedrigen arithmetischen Intensität eine deutlich verbesserte Performance erzielen. Für die Generierung dieser optimierten Implementierungsvarianten verschmelzen wir per Kernelfusion im Wesentlichen jeweils sämtliche Vektorgrundoperationen in jedem RHS $\rightarrow$ LC-Glied der Abhängigkeitskette des ODE-Verfahrens zu einem verschmolzenen Kernel. Ein solches Kernel kann nun die von ihm produzierten Vektorkomponenten zwischen den verschmolzenen Vektorgrundoperationen über den On-Chip-Speicher des Prozessors transferieren und muss diese nicht unnötig in den DRAM zurückzuschreiben. Zusätzlich muss das Kernel mehrmals benötigte Vektorkomponenten nur einmal aus dem DRAM in den On-Chip-Speicher laden, und kann diese Vektorkomponenten anschließend über den On-Chip-Speicher wiederverwenden. Ebenso zeigten wir, dass sich über die zusätzlichen Enabling-Transformationen die Datenwiederverwendung weiter erhöhen und die Performance weiter steigern lässt.

Um sowohl auf GPUs als auch auf CPUs optimierten Code für die fusionierten Kernels zu generieren, entwickelten wir für jede Architektur jeweils einen Code-Generator, welcher auf die Besonderheiten der jeweiligen Architektur eingeht. Der GPU-Codegenerator übersetzt dabei wieder jedes fusionierte Kernel in ein GPU-Kernel. Dabei zielt der Code des GPU-Kernels darauf ab, möglichst viele Daten über den Registersatz der GPU wiederzuverwenden und durch eine zur RHS-Funktion passende Kerneldimension möglichst lokal auf die permanenten Argumentvektoren der RHS-Operationen zuzugreifen. Im Gegensatz dazu übersetzt der CPU-Codegenerator wieder jedes fusionierte Kernel in ein paralleles Schleifennest. Dabei zielt der Code eines solchen CPU-Kernels darauf ab, möglichst viele Daten über die Caches der CPU wiederzuverwenden. Um die Datenwiederverwendung über die unteren Ebenen der Cache-Hierarchie zu forcieren, implementierten wir über unseren CPU-Codegenerator zwei unterschiedliche Schleifenstrukturen für die CPU-Kernel. Ebenso können die CPU-Kernels den Fetch-On-Write-Overhead des Last-Level-Caches der CPU vermeiden, indem sie mit Non-Temporal-Store-Instruktionen die von ihnen produzierten Vektorkomponenten in die permanenten Vektoren zurückschreiben.

Die anschließend durchgeführten Experimente mit den per Kernelfusion und Enabling-Transformationen generierten Varianten sowie mit BRUSS2D als Testproblem zeigten sowohl auf den untersuchten GPUs als auch auf den untersuchten CPUs ein sehr ähnliches Gesamtbild. So beträgt auf den getesteten GPUs der Speedup beim Verner-Verfahren zwischen 1.62 und 1.94 und beim PIRK-Verfahren zwischen 2.34 und 3.51, während auf den getesteten CPUs der Speedup beim Verner-Verfahren zwischen 1.98 und 2.15 und beim PIRK-Verfahren zwischen 4.14 und 4.15 beträgt. Dieser ähnlich große Speedup auf diesen beiden unterschiedlichen Prozessorarchitekturen ist darauf zurückzuführen, dass auf diesen beiden Prozessorarchitekturen die generierten Implementierungsvarianten meist stark durch die DRAM-Bandbreite gebunden sind. Dadurch können sie auf beiden Prozessorarchitekturen die Rechenleistung des jeweiligen Prozessors nur schlecht auslasten. So beträgt zum Beispiel auf der Maxwell die Auslastung der IPC je nach Variante und Verfahren zwischen 5 % und 15 % und die Auslastung der DRAM-Bandbreite zwischen 75 % und 87 %. Ebenso beträgt zum Beispiel auf dem Skylake die Ausnutzung der DP-Peak-Performance je nach Variante und Verfahren zwischen 1.2 % und 3.5 %

und die Auslastung der DRAM-Bandbreite zwischen 73 % und 82 %. Da auf allen untersuchten Architekturen und für sämtliche betrachteten Verfahren die Cache-Effizienz zwischen 92 % und 100 % beträgt, können die Caches weiterhin effizient das DRAM-Volumen beinahe auf das Datenvolumen reduzieren. Dadurch und auf Grund der zuvor genannten Limitierung durch die DRAM-Bandbreite ist die Laufzeit einer per Kernelfusion generierten Implementierungsvariante auf allen betrachteten GPUs und CPUs in etwa direkt proportional zu deren Datenvolumen.

10 Tiling

10.1 Motivation

Wie die Messungen im letzten Kapitel zeigten, so sind leider die per Kernelfusion generierten Varianten bei Anfangswertproblemen mit niedriger arithmetischer Intensität immer noch durch die DRAM-Bandbreite gebunden. Zusätzlich verhindert bei der allgemeinen Klasse von Anfangswertproblemen, die wir bis zu diesem Kapitel betrachteten, das beliebige Zugriffsmuster einer RHS-Operation auf ihren Argumentvektor die Fusion von LC $\rightarrow$ RHS-Abhängigkeiten. Folglich ist keine Datenwiederverwendung entlang dieser LC $\rightarrow$ RHS-Abhängigkeiten möglich, und wir können bei einem allgemeinen Anfangswertproblem keine weiteren Lokalitätsoptimierungen mehr durchführen. Allerdings können wir die Lokalität dennoch weiter erhöhen, indem wir uns auf eine bestimmte Klasse von Anfangswertproblemen spezialisieren. In dieser Arbeit spezialisieren wir uns auf Anfangswertprobleme mit beschränkter Zugriffsdistanz, da diese viele interessante Anfangswertprobleme, wie zum Beispiel Stencils, abdecken und auch die Datenwiederverwendung entlang einer LC $\rightarrow$ RHS-Abhängigkeit ermöglichen.

10.2 Theorie

10.2.1 Überblick

Selbst die Spezialisierung auf Anfangswertprobleme mit beschränkter Zugriffsdistanz erlaubt es uns nicht, eine LC $\rightarrow$ RHS-Abhängigkeit zu einem einzigen Kernel zu verschmelzen. Jedoch können wir eine solche LC $\rightarrow$ RHS-Abhängigkeit mithilfe einer *partitionierten Fusion* zu einem Kernelpaar oder Kerneltriplet verschmelzen. Allerdings muss diese partitionierte Fusion wiederum nach besonderen Regeln erfolgen.

Über die mehrmalige Anwendung dieser partitionierten Fusion entlang der RHS $\rightarrow$ LC-Abhängigkeitskette eines ODE-Verfahrens lassen sich zweidimensionale *Tiling-Schemata* verwirklichen, welche den Iterationsraum, der von der Systemdimension als räumliche Dimension und der RHS $\rightarrow$ LC-Abhängigkeitskette als zeitliche Dimension aufgespannt wird, mit *Tiles* (Kacheln) auffüllen. Als zweidimensionale Tiling-Schemata realisieren wir das *trapezoidale Tiling-Schema* und das *hexagonale Tiling-Schema*, welche den Iterationsraum mit *Trapezoiden* beziehungsweise mit *Hexagonen* als Tiles auffüllen. Bei diesen beiden Tiling-Schemata verläuft die *Breite* eines Tiles entlang der Systemdimension, während die *Höhe* eines Tiles entlang der zeitlichen Dimension beziehungsweise entlang der RHS $\rightarrow$ LC-Abhängigkeitskette verläuft. Dabei sind die Tiles entlang der Systemdimension unabhängig voneinander, wodurch sie parallel zueinander von den Rechenkernen des Prozessors berechnet werden können. Innerhalb eines solchen Tiles kann ein Prozessor seine On-Chip-Speicher verwenden, um die vom Tile produzierten Vektorkomponenten zu transferieren oder ein redundantes Laden von Vektorkomponenten aus dem DRAM zu vermeiden. Zusätzlich muss ein Tile nur diejenigen produzierten Vektorkomponenten in den DRAM zurückschreiben, die auch von einem nachfolgenden Tile gelesen werden. Dadurch kann das Tiling gegenüber der Kernelfusion die Datenwiederverwendung weiter erhöhen.

Nachdem wir für ein ODE-Verfahren ein Tiling durch die mehrmalige Anwendung der partitionierten Fusion über Kernels definiert haben, erzeugen wir als Nächstes aus jedem dieser Kernels über einen Codegenerator ein ausführbares Programm für den Zielrechner, welches

speziell auf die Architektur dieses Zielrechners optimiert ist. Danach können wir mit der Zeitschrittprozedur beginnen, welche die Zeitschritte des ODE-Verfahrens auf dem Zielrechner berechnet. Diese Zeitschrittprozedur führt wiederum für jeden Zeitschritt die Kernels des Tilings beziehungsweise die ausführbaren Objekte im Datenflussgraphen gemäß ihrer topologischen Reihenfolge aus. Ein solches Kernel berechnet dabei, je nach Tiling-Schema, entweder eine Menge von nebenläufigen Trapezoiden oder eine Menge von nebenläufigen Hexagonen, wobei ein jedes Tile wiederum seine fusionierten Vektorgrundoperationen gemäß ihrer topologischen Ordnung berechnet.

Wir können beide Tiling-Schemata nicht nur über die Stufen eines ODE-Verfahrens verlaufen lassen, was wir im Folgenden als *Tiling über die Stufen* bezeichnen, sondern zusätzlich auch über die Zeitschritte eines ODE-Verfahrens, was wir im Folgenden als *Tiling über die Zeitschritte* bezeichnen. Um dieses Tiling über die Zeitschritte zu beschreiben, müssen wir zunächst die Regeln für das Tiling erweitern. Zusätzlich müssen wir den Datenflussgraphen potentiell über eine besondere Enabling-Transformation, die sogenannte *Abroll-Transformation*, abrollen, so dass die Grundoperationen im abgerollten Datenflussgraphen nicht mehr einen einzigen Zeitschritt des ODE-Verfahrens, sondern mehrere aufeinanderfolgende Zeitschritte beschreiben. Zudem benötigen wir für das Tiling über die Zeitschritte eine spezielle *Prologiteration* und *Epilogiteration* mit speziellen *Prologkernels* und *Epilogkernels*.

Sowohl das trapezoidale als auch das hexagonale Tiling-Schema erfordern, dass wir die Tile-Breiten und die Tile-Höhen geschickt wählen. Da auf Grund der Freiheitsgrade in beiden Tiling-Schemata eine kombinatorische Explosion auftritt, wodurch wiederum eine sehr große Anzahl an möglichen Kombinationen für die Tile-Breiten und Tile-Höhen entsteht, stellen wir in diesem Kapitel einen Autotuning-Ansatz vor. Dieser Autotuning-Ansatz vermeidet die kombinatorische Explosion, indem er die Einflüsse von den Tile-Breiten und Tile-Höhen auf die Performance voneinander entkoppelt. Zusätzlich bestimmt dieser Ansatz für jedes Kernel einen optimalen Satz an internen Parametern, wie zum Beispiel die optimale interne Schleifenstruktur.

In der einfachsten Variante vom Tiling, dem *Single-Thread-Tiling* oder dem *Single-Core-Tiling*, wird ein jedes Tile jeweils nur von einem einzigen Thread oder Kern eines Prozessors bearbeitet, wodurch ein moderner Mehrkernprozessor viele nebenläufige Tiles benötigt, um komplett ausgelastet zu sein. Dadurch kann ein jedes Tile nur einen kleinen Anteil des On-Chip-Speichers des Prozessors belegen, wodurch wiederum die maximale Tile-Größe, welche noch in dem On-Chip-Speicher Platz findet, stark limitiert ist. Da allerdings größere Tiles, sofern sie im On-Chip-Speicher des Prozessors Platz finden, die Anzahl an DRAM-Zugriffen stärker reduzieren als kleinere Tiles, besitzen größere Tiles potentiell eine bessere Performance. Ein Ansatz, um den pro Tile verfügbaren On-Chip-Speicher im Vergleich zum Single-Thread-Tiling oder zum Single-Core-Tiling zu erhöhen, ist das *Multi-Core-Tiling*, bei welchem mehrere Kerne eines Prozessors an einem Tile zusammenarbeiten. Dadurch benötigt ein Prozessor auch weniger nebenläufige Tiles, um ausgelastet zu sein, und ein jedes Tile kann einen größeren Anteil des On-Chip-Speichers des Prozessors belegen. Somit kann das Multi-Core-Tiling die Zahl an DRAM-Zugriffen stärker reduzieren als das Single-Thread-Tiling und das Single-Core-Tiling. Dadurch lohnt sich das Multi-Core-Tiling noch für große Zugriffsdistanzen, während sich das Single-Core-Tiling oder das Single-Core-Tiling nur für kleine Zugriffsdistanzen lohnt.

10.2.2 Blockweise Aufteilung von Systemkomponenten auf die parallelen Blöcke eines Kernels

Für die systemparallele Berechnung von Vektorgrundoperationen beim Tiling weisen wir jedem Block eines Kernels ein Intervall von Komponenten für jede Vektorgrundoperation zu, so dass für (RHS | LC | MAP)-Operationen ein Block des Kernels w aufeinanderfolgende Vektorkomponenten berechnet und für RED-Operationen die partielle Reduktion von w aufeinanderfolgenden Vektorkomponenten durchführt. Dabei erlauben wir, dass eine Vektorgrundoperation von unterschiedlichen Kernels berechnet werden darf, wobei jedes Kernel eine Partition dieser Vektorgrundoperation berechnet. Zusätzlich erlauben wir, dass ein Block eines Kernels für jede der Vektorgrundoperationen des Kernels ein unterschiedliches Intervall an Komponenten berechnen darf.

Dies steht somit im Kontrast zu den Varianten mit Kernelfusion, bei welchen wir zwar erlauben, dass die zu berechnenden Komponenten eines Blocks nicht aufeinanderfolgend sein müssen, jedoch fordern, dass ein Kernel für alle seiner Vektorgrundoperationen das gleiche Mapping zwischen Vektorkomponenten und Blöcken besitzen muss, und dass eine Vektorgrundoperation nur von einem einzigen Kernel berechnet werden darf.

Um die Abhängigkeiten zwischen den Kernels eines Zeitschritts zu bestimmen, definieren wir fünf unterschiedliche Indexmengen für jeden Block i einer Vektorgrundoperation o eines Kernels k, wobei es sich bei jeder dieser Mengen um ein oder um mehrere Intervalle von Vektorkomponenten handelt:

- **Argumentintervall $\mathcal{A}(o,k,i)$:** Das *Argumentintervall* umfasst diejenige Menge an Komponenten der Vektorgrundoperation, die die Aufteilung dem Block des Kernels zur Berechnung zuweist.
- **Eingabeintervall $\mathcal{I}(o,k,i)$:** Das *Eingabeintervall* umfasst diejenige Menge an Komponenten der Eingabevektoren der Vektorgrundoperation, die der Block für die Berechnung seiner Komponenten als Eingabe benötigt.
- **Ausgabeintervall $\mathcal{O}(o,k,i)$:** Das *Ausgabeintervall* umfasst diejenige Menge an Komponenten des Ausgabevektors der Vektorgrundoperation, die der Block bei der Berechnung seiner Komponenten als Ausgabe produziert.
- **DRAM-Leseintervalle $\mathcal{R}(o,k,i)$:** Die *DRAM-Leseintervalle* umfassen diejenige Menge an Komponenten der Eingabevektoren, die der Block für die Berechnung seiner Komponenten aus den dazugehörigen permanenten Vektoren und damit aus dem DRAM lesen muss.
- **DRAM-Schreibintervalle $\mathcal{W}(o,k,i)$:** Die *DRAM-Schreibintervalle* umfassen diejenige Menge an Komponenten des Ausgabevektors der Vektorgrundoperation, die der Block in den dazugehörigen permanenten Vektor und damit in den DRAM zurückschreiben muss.

Für jede Vektorgrundoperation hängt das Eingabeintervall und das Ausgabeintervall von dem Argumentintervall ab. Diese Intervalle sind in Tabelle 10.1 übersichtlich zusammengefasst. Als einen Ausgangspunkt für folgende Betrachtungen zeigt die Tabelle 10.2 diejenigen Eingabe- und Ausgabeintervalle für die verschiedenen Vektorgrundoperationen, welche daraus resultieren, dass ein Kernel für jede Vektorgrundoperation gestartet wird und jedem Block des Kernels

Op. o	$\mathcal{A}(o,k,i)$	$\mathcal{I}(o,k,i)$	$\mathcal{O}(o,k,i)$
RHS	[first, last]	[first $-$ d, last $+$ d]	[first, last]
LC	[first, last]	[first, last]	[first, last]
MAP	[first, last]	[first, last]	[first, last]
RED	[first, last]	[first, last]	n/a

Tabelle 10.1: Eingabe- und Ausgabeintervalle der Vektorgrundoperationen für ein gegebenes Argumentintervall. Hierbei beschreibt *first* die erste Komponente im Argumentinervall und *last* die letzte Komponente.

Op. o	$\mathcal{A}(o,k,i)$	$\mathcal{I}(o,k,i)$	$\mathcal{O}(o,k,i)$
RHS	$[w\cdot i, w\cdot i+w-1]$	$[w\cdot i-d, w\cdot i+w-1+d]$	$[w\cdot i, w\cdot i+w-1]$
LC	$[w\cdot i, w\cdot i+w-1]$	$[w \cdot i, w \cdot i + w - 1]$	$[w\cdot i, w\cdot i+w-1]$
MAP	$[w\cdot i, w\cdot i+w-1]$	$[w \cdot i, w \cdot i + w - 1]$	$[w\cdot i, w\cdot i+w-1]$
RED	$[w\cdot i, w\cdot i+w-1]$	$[w \cdot i, w \cdot i + w - 1]$	n/a

Tabelle 10.2: Blockweise Aufteilung von Vektorgrundoperation auf die Blöcke eines Kernels. In dieser Tabelle wird die Blockgröße der Aufteilung mit w bezeichnet.

jeweils ein Intervall von w aufeinanderfolgenden Vektorkomponenten zur Berechnung zugewiesen wird.

Offensichtlich sind bei (RHS | LC | MAP)-Operationen das Argumentintervall und das Ausgabeintervall identisch. Nichtsdestotrotz ist die Unterscheidung zwischen dem Argumentintervall und dem Ausgabeintervall bei RED-Operationen nützlich. Denn bei RED-Operationen beschreibt das Argumentintervall diejenige Menge an Komponenten, die der Block des Kernels partiell reduziert, während das Ausgabeintervall bei einer RED-Operation nicht definiert ist, da eine RED-Operation als Ergebnis keinen Vektor, sondern ein Skalar produziert. Des Weiteren ist für RHS-Operationen bei einem Anfangswertproblem mit beschränkter Zugriffsdistanz das Eingabeintervall an der rechten und linken Intervallgrenze um jeweils die beschränkte Zugriffsdistanz größer als das Argumentintervall, das heißt $\mathcal{I}(o,k,i) = \mathcal{A}(o,k,i) + [-d,d]$. Im Gegensatz dazu müssen wir bei den allgemeinen Anfangswertproblemen, die wir in den vorherigen Kapiteln betrachteten, ein Eingabeintervall von $\mathcal{I}(o,k,i) = [1,n]$ für RHS-Operationen annehmen.

10.2.3 Intervallbasierte Fusionsregeln

Falls wir, um ein Tiling zu erzielen, Partitionen von mehreren Vektorgrundoperationen vom selben Kernel berechnen lassen wollen, so dürfen wir die Argumentintervalle für diese Vektorgrundoperationen nach bestimmten Regeln wählen. Diese Regeln setzen sich, wie bei der Kernelfusion, zusammen aus den allgemeinen Voraussetzungen, die die gewählten Argument-

intervalle erfüllen müssen, und weitere von diesen Voraussetzungen und dem Speicherzugriffsmuster der Vektorgrundoperationen abgeleitete Regeln, welche Argumentintervalle für zwei voneinander abhängige Vektorgrundoperationen valide sind.

In dieser Arbeit müssen die gewählten Argumentintervalle folgende Voraussetzungen erfüllen:

i) Jede Komponente einer Vektorgrundoperation muss von dem Argumentintervall von genau einem Block abgedeckt werden. Daraus folgt auch, dass sich die Argumentintervalle nicht überlappen dürfen.

   Anmerkung: Überlappende Argumentintervalle können auf Kosten von redundanten Berechnungen ebenfalls die Datenwiederverwendung erhöhen. Diesen Ansatz werden wir in dieser Arbeit aber nicht weiter verfolgen.

ii) Diejenigen Blöcke, die eine Vektorgrundoperation berechnen, dürfen zu unterschiedlichen Kernels gehören.

iii) Der Prozessor führt die Kernels eins nach dem anderen aus. Deshalb dürfen die gewählten Argumentintervalle keine zyklischen Abhängigkeiten zwischen den Kernels verursachen.

iv) Die gewählten Argumentintervalle dürfen keine Abhängigkeiten zwischen den Blöcken eines Kernels verursachen, weil der Prozessor nicht alle Blöcke des Kernels parallel zueinander ausführen kann.

Aus diesen vier Voraussetzungen können wir die folgenden formalen Regeln für die Validität der Fusion von zwei Vektorgrundoperationen a und b zu einem Kernel k ableiten:

1. **Keine Abhängigkeit zwischen a und b:** Beliebige, sich nicht überlappende Argumentintervalle sind erlaubt.
2. **Direkte Abhängigkeit von a nach b ($a \to b$):** Die Validität der Fusion hängt von dem Eingabe- und Ausgabeintervall der Blöcke des Kernels ab:
   a) $\mathcal{O}(a,k,i) \supseteq \mathcal{I}(b,k,i)$ für alle Blöcke i von Kernel k: Die Fusion von a und b ist erlaubt, weil der $i$te Block alle Komponenten von a produziert, die er benötigt, um seine zugewiesenen Komponenten von b zu berechnen. Folglich gibt es keine Abhängigkeiten zwischen den Blöcken des Kernels.
   b) $\mathcal{O}(a,k,i) \subset \mathcal{I}(b,k,i)$ für zumindest einen Block i des Kernels k: Wegen der Voraussetzung iv ist die Fusion von a und b nur dann erlaubt, falls die fehlenden Komponenten $\mathcal{I}(b,k,i) \setminus \mathcal{O}(a,k,i)$ (diejenigen Komponenten, die nicht von dem $i$ten Block von a produziert werden, aber von dem $i$ten Block von b benötigt werden) von einem anderen Kernel $k' \neq k$ erzeugt werden.
   c) $\mathcal{O}(a,k,i)$ ist nicht definiert (a ist eine RED-Operation): Dieser Fall kann nicht auftreten, da eine andere Vektorgrundoperation nicht direkt von einer RED-Operation abhängen kann.
3. **Indirekte Abhängigkeit von a nach b (zum Beispiel eine Abhängigkeitskette $a \to o_1 \to o_2 \to \cdots \to b$):** Wegen der Voraussetzung iii) muss ein fusioniertes Kernel mit a und b auch alle Vektorgrundoperationen, die die Abhängigkeit von a nach b tragen, fusionieren (also $o_1, o_2, \ldots$ im soeben genannten Beispiel). Ob alle benötigten Abhängigkeiten fusioniert werden dürfen, wird für jede direkte Abhängigkeit, zum Beispiel $a \to o_1$, $o_1 \to o_2$, durch die Regel 2 bestimmt.

10.2.4 Fusion von (RHS | LC | MAP) $\rightarrow$ (LC | MAP | RED)-Abhängigkeiten zu einem Kernel

Die Regeln aus dem vorherigen Abschnitt erlauben es direkt (RHS | LC | MAP) $\rightarrow$ (LC | MAP | RED)-Abhängigkeiten zu einem einzigen Kernel zu verschmelzen. Hierfür müssen beide Vektorgrundoperationen das gleiche Mapping zwischen Vektorkomponenten und Blöcken des Kernels, beziehungsweise die gleichen Argumentintervalle besitzen, wie es von Tabelle 10.2 gezeigt wird. Falls wir eine blockweise Aufteilung von Vektorkomponenten auf die Blöcke des Kernels mit einer Blockgröße von w verwenden, resultiert dies in einem Argumentintervall von $[w \cdot i, w \cdot i + w - 1]$. Dadurch hat der $i$te Block der (RHS | LC | MAP)-Operationen als Startoperation der Abhängigkeit das Ausgabeintervall von $[w \cdot i, w \cdot i + w - 1]$. Dieses Ausgabeintervall ist identisch zu dem Eingabeintervall des $i$ten Blocks der (LC | MAP | RED)-Operation als Endoperation der Abhängigkeit. Folglich verursacht die Fusion der Blöcke der Startoperation und Endoperation der (RHS | LC | MAP) $\rightarrow$ (LC | MAP | RED)-Abhängigkeit keine Abhängigkeiten zwischen den Blöcken des resultierenden fusionierten Kernels, wodurch sie erlaubt ist. Diese Art der Verschmelzung zweier Vektorgrundoperationen ist weitestgehend analog zu der Kernelfusion, wie wir sie im 9. Kapitel durchführten, mit demjenigen Unterschied, dass wir nun voraussetzen, dass ein Block eines Kernels für sämtliche verschmolzenen Vektorgrundoperationen jeweils einen Block von aufeinanderfolgenden Komponenten berechnet.

10.2.5 Unmöglichkeit der Fusion von LC $\rightarrow$ RHS-Abhängigkeiten zu einem Kernel

Jedoch ist gemäß der Voraussetzung i), das heißt ohne redundante Berechnungen, eine LC $\rightarrow$ RHS-Abhängigkeit nicht zu einem einzigen Kernel verschmelzbar. Denn bei einer Verschmelzung würde zumindest für einen Teil der Blöcke ein Block bei der Auswertung der LC-Operation nur eine Teilmenge derjenigen Komponenten produzieren, die der Block für die Berechnung seiner Komponenten der RHS-Operation benötigen würde. So hätte zum Beispiel im Falle einer blockweisen Aufteilung mit einer Blockgröße von w die LC-Operation eines Blockes i ein Ausgabeintervall von $[w \cdot i, w \cdot i + w - 1]$, während die RHS-Operation dieses Blockes i ein Eingabeintervall von $[w \cdot i - d, w \cdot i + w - 1 + d]$ hätte. Dieses Eingabeintervall der RHS-Operation wäre somit um $2d$ Komponenten größer als das Ausgabeintervall der LC-Operation. Deshalb ist die Fusion einer LC $\rightarrow$ RHS-Abhängigkeit zu einem einzigen Kernel gemäß der Regel 2b) nicht erlaubt, da diese Fusion Abhängigkeiten zwischen den Blöcken des Kernels verursachen würde.

10.2.6 Partitionierte Fusion

Glücklicherweise verbietet die Regel 2b) nicht die Fusion von LC $\rightarrow$ RHS-Abhängigkeiten allgemein, sondern sie setzt voraus, dass im Falle von $\mathcal{O}(\text{LC}, k, i) \subset \mathcal{I}(\text{RHS}, k, i)$ die fehlenden Komponenten des Argumentvektors der RHS-Operation durch ein anderes Kernel erzeugt werden müssen. Während es somit nicht erlaubt ist, eine LC $\rightarrow$ RHS-Abhängigkeit direkt zu einem einzigen Kernel zu verschmelzen, so ist es jedoch möglich, Partitionen, das heißt Blöcke von aufeinanderfolgenden Komponenten, dieser beiden Vektorgrundoperationen zu verschmelzen. Diese Partitionen können dann entweder zyklisch auf zwei unterschiedliche Kernel k_1 und k_2 aufgeteilt werden, welche wir im folgenden als *partitionierte Fusion von Typ-PAIR* bezeichnen, oder auf drei Kernel k_1, k_2 und k_3, die wir im folgenden als *partitionierte Fusion vom Typ-TRIP* bezeichnen.

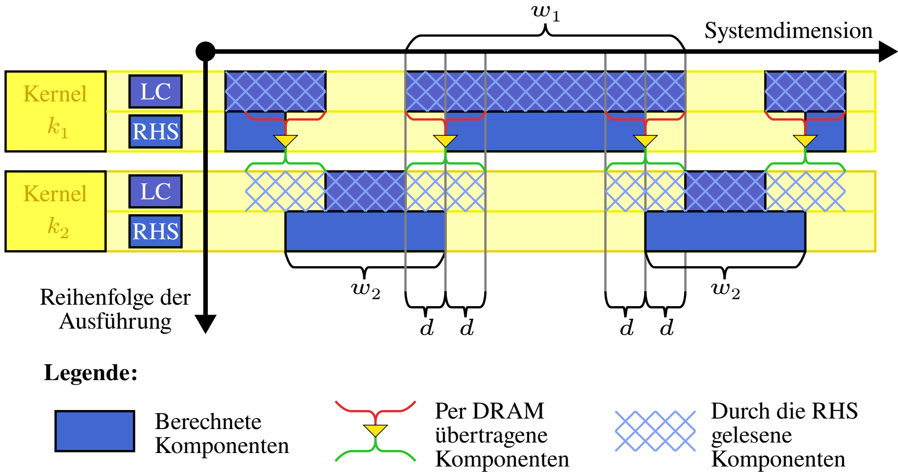


Abbildung 10.1: Partitionierte Fusion einer LC $\rightarrow$ RHS-Abhängigkeit vom Typ-PAIR.

10.2.6.1 Partitionierte Fusion vom Typ-PAIR: Fusion zu einem Kernelpaar

Die *partitionierte Fusion vom Typ-PAIR* verarbeitet die beiden Vektorgrundoperationen in zwei Phasen, die jeweils einem Kernel k_1 und k_2 entsprechen. In jeder Phase wählt diese partitionierte Fusion vom Typ-PAIR die Argumentintervalle von den Vektorgrundoperationen in Übereinstimmung mit den Fusionsregeln, die wir im Abschnitt 10.2.3 ableiteten, wie folgt (siehe auch Abbildung 10.1 und Tabelle 10.3):

- **Phase 1 (k_1):** Die partitionierte Fusion vom Typ-PAIR wählt die Argumentintervalle für die Blöcke des Kernels k_1 zunächst so, dass $\mathcal{O}(\text{LC}, k_1, i) \supseteq \mathcal{I}(\text{RHS}, k_1, i)$ (Regel 2a) gilt. Dadurch berechnet ein Block i des Kernels k_1 all diejenigen Komponenten der LC-Operation, die er für die darauffolgende Berechnung der RHS-Operation benötigt. Dadurch ist das maximale Argumentintervall der RHS-Operation $\mathcal{A}(\text{RHS}, k_1, i) = \mathcal{O}(\text{LC}, k_1, i) - [-d, +d]$.

- **Phase 2 (k_2):** Das Kernel k_2 kann nun die Partitionen der LC-Operation und RHS-Operation mit $\mathcal{O}(\text{LC}, k_2, i) \not\supseteq \mathcal{I}(\text{RHS}, k_2, i)$ berechnen. Jedoch muss die partitionierte Fusion vom Typ-PAIR die Argumentintervalle im Kernel k_2 wegen der Regel 2b so wählen, dass $\mathcal{O}(\text{LC}, k_2, i) \cup \mathcal{O}(\text{LC}, k_1, i) \cup \mathcal{O}(\text{LC}, k_1, i+1) \supseteq \mathcal{I}(\text{RHS}, k_2, i)$ gilt. Da für die Blöcke in dem Kernel k_1 das Argumentintervall der RHS-Operation um mindestens $2d$ Komponenten kleiner sein muss als das Argumentintervall der LC-Operation, und das Kernel k_2 alle verbleibenden Vektorkomponenten von beiden Vektorgrundoperationen abdecken muss, muss ein Block von dem Kernel k_2 ein größeres Argumentintervall für die RHS-Operation als für die LC-Operation abdecken. Dadurch muss für das minimale Argumentintervall der RHS-Operation $\mathcal{A}(\text{RHS}, k_2, i) = \mathcal{O}(\text{LC}, k_1, i) + [-d, +d]$ gelten.

Argumentintervall und das resultierende Eingabe- und Ausgabeintervall beim Typ-PAIR

Kernel	Op. o	$\mathcal{A}(o,k,i)$	$\mathcal{I}(o,k,i)$	$\mathcal{O}(o,k,i)$
k_1	LC	$[p\cdot i, p\cdot i + w_1 - 1]$	$\mathcal{A}(\text{LC}, k_1, i)$	$\mathcal{A}(\text{LC}, k_1, i)$
	RHS	$\mathcal{A}(\text{LC}, k_1, i) - [-d, +d]$	$\mathcal{A}(\text{LC}, k_1, i)$	$\mathcal{A}(\text{RHS}, k_1, i)$
k_2	LC	$\mathcal{A}(\text{LC}, k_1, i) + [w_1, +(w_2 - 2d)]$	$\mathcal{A}(\text{LC}, k_2, i)$	$\mathcal{A}(\text{LC}, k_2, i)$
	RHS	$\mathcal{A}(\text{LC}, k_2, i) + [-d, +d]$	$\mathcal{A}(\text{RHS}, k_2, i) + [-d, +d]$	$\mathcal{A}(\text{RHS}, k_2, i)$

Über den DRAM übertragene Vektorkomponenten beim Typ-PAIR

Kernel	Op. o	$\mathcal{R}(o,k,i)$	$\mathcal{W}(o,k,i)$
k_1	LC	$\mathcal{I}(\text{LC}, k_1, i)$	$[p\cdot i, p\cdot i + 2d - 1] \cup [p\cdot i + w_1 - 2d, p\cdot i + w_1 - 1]$
	RHS	$\varnothing$	$\mathcal{O}(\text{RHS}, k_1, i)$
k_2	LC	$\mathcal{I}(\text{LC}, k_2, i)$	$\varnothing$
	RHS	$[p\cdot i + w_1 - 2d, p\cdot i + w_1 - 1] \cup [p(i+1), p(i+1) + 2d - 1]$	$\mathcal{O}(\text{RHS}, k_2, i)$

Tabelle 10.3: Blockzyklische Datenverteilung bei einer verteilten Fusion vom Typ-PAIR. In diesen Tabellen ist p die Periode der blockzyklischen Aufteilung, das heißt $p = w_1 + w_2 - 2d$, und w_1 sowie w_2 sind jeweils die Tile-Breite eines Kernels, das heißt das maximale Argumentintervall über alle Vektorgrundoperationen des Kernels k_1 beziehungsweise k_2.

Bei einer partitionierten Fusion vom Typ-PAIR kann ein Block i vom Kernel k_2 die RHS-Operation für das Argumentintervall $\mathcal{A}(\text{LC}, k_2, i) - [-d, +d]$ nur mit denjenigen Komponenten der LC-Operation berechnen, die er selbst produziert hat, und muss die Komponenten $\mathcal{I}(\text{RHS}, k_2, i) \setminus \mathcal{O}(\text{LC}, k_2, i)$ der LC-Operation aus deren permanenten Vektor und damit aus dem DRAM lesen, weshalb das Kernel k_1 nur ebendiese Komponenten der LC-Operation in den permanenten Vektor und damit in den DRAM schreiben muss.

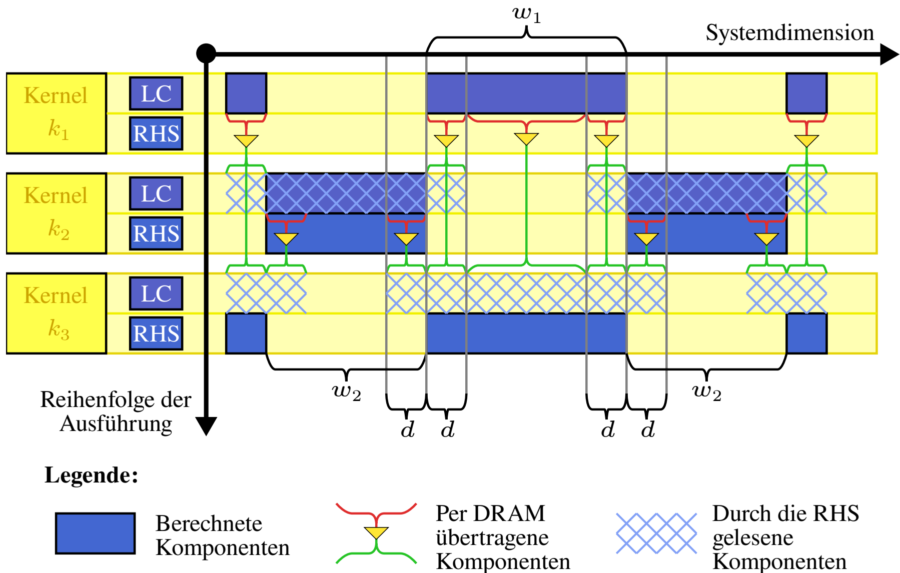


Abbildung 10.2: Partitionierte Fusion einer LC → RHS-Abhängigkeit vom Typ-TRIP.

10.2.6.2 Partitionierte Fusion vom Typ-TRIP: Fusion zu einem Kerneltriplet

Im Fall von der *partitionierten Fusion vom Typ-TRIP* gibt es drei Phasen (vergleiche Abbildung 10.2 und Tabelle 10.4), wobei jede Phase einem der folgenden drei Kernels entspricht:

- **Phase 1 (k_1):** Die Blöcke von Kernel k_1 berechnen nur die Partitionen der LC-Operation, aber nicht der RHS-Operation.
- **Phase 2 (k_2):** Die Blöcke von Kernel k_2 berechnen die verbleibenden Partitionen der LC-Operationen, welche nicht durch das Kernel k_1 berechnet wurden, und berechnen die Partitionen der RHS-Operation für das jeweils das gleiche Argumentintervall.
- **Phase 3 (k_3):** Die Blöcke des Kernels k_3 berechnen die verbleibenden Partitionen der RHS-Operation, aber keine Partitionen der LC-Operation.

Bei der partitionierten Fusion vom Typ-TRIP benötigt das Kernel k_2 für die Auswertung seiner RHS-Operation zusätzlich noch jeweils die Komponenten am rechten und linken Rand von jedem Block des Kernels k_1, welche es somit aus dem permanenten Vektor und damit aus DRAM laden muss. Ebenso benötigt das Kernel k_3 für die Auswertung seiner RHS-Operation sämtliche Komponenten der LC-Operation, die das Kernel k_1 produziert hat, und zusätzlich noch jeweils die Komponenten am rechten und linken Rand von jedem Block des Kernels k_2, die es ebenfalls alle aus dem permanenten Vektor und damit aus den DRAM laden muss. Deshalb muss das Kernel k_1 sämtliche berechneten Komponenten der LC-Operation in deren permanenten

Argumentintervall und das resultierende Eingabe- und Ausgabeintervall beim Typ-TRIP

Kernel	Op. o	$\mathcal{A}(o,k,i)$	$\mathcal{I}(o,k,i)$	$\mathcal{O}(o,k,i)$
k_1	LC	$[p \cdot i, p \cdot i + w_1 - 1]$	$\mathcal{A}(\text{LC}, k_1, i)$	$\mathcal{A}(\text{LC}, k_1, i)$
	RHS	$\varnothing$	$\varnothing$	$\varnothing$
k_2	LC	$\mathcal{A}(\text{LC}, k_1, i)+$ $[w_1, +w_2]$	$\mathcal{A}(\text{LC}, k_2, i)$	$\mathcal{A}(\text{LC}, k_2, i)$
	RHS	$\mathcal{A}(\text{LC}, k_2, i)$	$\mathcal{A}(\text{RHS}, k_2, i)+$ $[-d, +d]$	$\mathcal{A}(\text{RHS}, k_2, i)$
k_3	LC	$\varnothing$	$\varnothing$	$\varnothing$
	RHS	$\mathcal{A}(\text{LC}, k_1, i)$	$\mathcal{A}(\text{RHS}, k_3, i)+$ $[-d, +d]$	$\mathcal{A}(\text{RHS}, k_3, i)$

Über den DRAM übertragene Vektorkomponenten beim Typ-TRIP

Kernel	Op. o	$\mathcal{R}(o,k,i)$	$\mathcal{W}(o,k,i)$
k_1	LC	$\mathcal{I}(\text{LC}, k_1, i)$	$\mathcal{O}(\text{LC}, k_1, i)$
	RHS	$\varnothing$	$\varnothing$
k_2	LC	$\mathcal{I}(\text{LC}, k_2, i)$	$\varnothing$
	RHS	$[pi + w_1 - 2d, pi + w_1 - 1] \cup$ $[p(i+1), p(i+1) + 2d - 1]$	$\mathcal{O}(\text{RHS}, k_2, i)$
k_3	LC	$\varnothing$	$\varnothing$
	RHS	$\mathcal{I}(\text{RHS}, k_3, i)$	$\mathcal{O}(\text{RHS}, k_3, i)$

Tabelle 10.4: Blockzyklische Datenverteilung bei einer verteilten Fusion vom Typ-TRIP. Dabei ist p die Periode der blockzyklischen Aufteilung, das heißt $p = w_1 + w_2 - 2d$, und w_1 sowie w_2 sind jeweils die Tile-Breite eines Kernels, das heißt das maximale Argumentintervall über alle Vektorgrundoperationen des Kernels k_1 beziehungsweise k_2.

Vektor und damit in den DRAM zurückschreiben, während das Kernel k_2 nur die Komponenten der LC-Operation am rechten und linken Rand eines jeden Blockes in den permanenten Vektor und damit in den DRAM zurückschreiben muss. Somit besitzt eine einzige partitionierte Fusion vom Typ-TRIP eine schlechtere Datenwiederverwendung als eine partitionierte Fusion vom Typ-PAIR. Jedoch ist eine partitionierte Fusion vom Typ-TRIP nützlich, um bei dem hexagonalen Tiling-Schema die Datenwiederverwendung zu erhöhen.

10.2.7 Tiling Schemata

10.2.7.1 Grundidee

Wie wir bereits in dem Abschnitt 6.6 erwähnten, so beinhaltet die transitive Reduktion des Datenflussgraphen von vielen ODE-Verfahren eine Abhängigkeitskette von RHS $\rightarrow$ LC-Gliedern. Dies können wir für die Maximierung der Datenwiederverwendung ausnutzen, indem wir die partitionierte Fusion mehrmals entlang dieser Abhängigkeitskette anwenden, so dass ein Kernel Partitionen vieler Vektorgrundoperationen berechnet und dementsprechend viele Daten zwischen diesen Vektorgrundoperationen wiederverwenden kann. Dieses mehrmalige Anwenden der partitionierten Fusion führt wiederum zu zwei zweidimensionalen Tiling-Schemata, und zwar dem *hexagonalen Tiling-Schema* und dem *trapezoidalen Tiling-Schema* (siehe Abbildung 10.4), bei welchen die eine Dimension die Systemdimension und die andere Dimension die Zeitdimension ist. Im Folgenden werden wir die maximale Größe eines Tiles entlang der Systemdimension als dessen *Tile-Breite* w bezeichnen. Zudem werden wir die Anzahl der in einem Tile verschmolzenen Glieder der Abhängigkeitskette als dessen *Tile-Höhe* h bezeichnen. Um die beste Performance zu erreichen, müssen wir die Tile-Breite und Tile-Höhe spezifisch auf das gewählte Anfangswertprobleme mit seiner beschränkten Zugriffsdistanz, das ODE-Verfahren und den benutzten Prozessor abstimmen.

10.2.7.2 Trapezoidales Tiling-Schema

Das einfachere *trapezoidale Tiling-Schema* wendet die partitionierte Fusion vom Typ-PAIR auf eine direkte Art iterativ an, so dass für eine ausgewählte Anzahl von RHS $\rightarrow$ LC-Gliedern das Argumentintervall der Grundoperationen von dem Kernel k_1 entlang der Abhängigkeitskette schrumpft, während das Argumentintervall der Grundoperationen von dem Kernel k_2 wächst. Somit bilden die Tiles dieser beiden Kernels entlang der Systemdimension eine Zeile aus Trapezoiden. Nach dieser ausgewählten Anzahl von RHS $\rightarrow$ LC-Gliedern startet das trapezoidale Tiling-Schema zwei weitere Kernels, k_3 und k_4, die zusammen eine weitere Zeile von Trapezoiden berechnen. Dies lässt sich weiter fortsetzen, bis das Tiling entlang der kompletten Abhängigkeitskette verläuft.

Um die Tile-Höhen und die Tile-Breiten für das trapezoidale Tiling zu wählen, existieren folgende Freiheitsgrade und Zwangsbedingungen: Alle Trapezoide in einer Zeile müssen gleich hoch sein, während Trapezoide in unterschiedlichen Zeilen unterschiedlich hoch sein dürfen. Des Weiteren müssen in einer Zeile alle ungeraden Trapezoide gleich breit sein, und alle geraden Trapezoide müssen ebenfalls die gleich breit sein. Dahingegen dürfen Trapezoide aus unterschiedlichen Zeilen unterschiedlich breit sein. Zudem muss für die Breite w und für die Höhe h eines Trapezoids folgende Ungleichung erfüllt sein:

$$w - 2d\,(h-1) \geq d \tag{10.1}$$

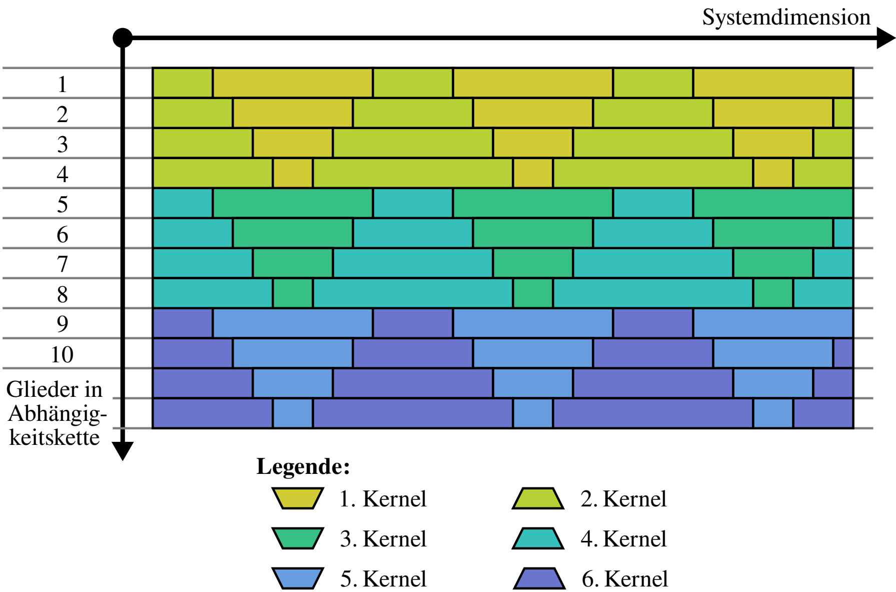


Abbildung 10.3: Trapezoidales Tiling-Schema

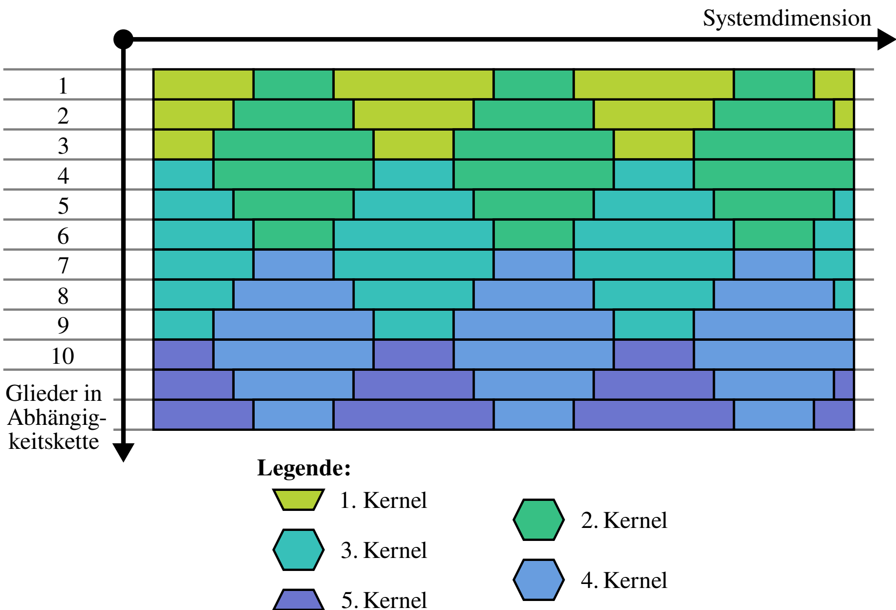


Abbildung 10.4: Hexagonales Tiling-Schema

10.2.7.3 Hexagonales Tiling-Schema

Das *hexagonale Tiling-Schema* wendet zunächst ebenfalls die partitionierte Fusion vom Typ-PAIR auf eine direkte Art iterativ an, so dass für eine ausgewählte Anzahl von RHS $\rightarrow$ LC-Gliedern das Argumentintervall der Grundoperationen von dem Kernel k_1 entlang der Abhängigkeitskette schrumpft, während das Argumentintervall der Grundoperationen von dem Kernel k_2 wächst. Nach diesen RHS $\rightarrow$ LC-Gliedern wendet das hexagonale Tiling-Schema allerdings die partitionierte Fusion vom Typ-TRIP an, um das Kernel k_1 durch ein drittes Kernel k_3 zu ersetzen. Anschließend wendet das hexagonale Tiling-Schema wiederum die partitionierte Fusion vom Typ-PAIR entlang der Abhängigkeitskette an, wobei es nun das Argumentintervall vom Kernel k_2 schrumpfen lässt, während es das Argumentintervall vom Kernel k_3 wachsen lässt. Dies lässt sich wieder weiter fortsetzen, bis das hexagonale Tiling entlang der kompletten Abhängigkeitskette verläuft. Dadurch bilden sich beim hexagonalen Tiling-Schema senkrecht zu der Zeitdimension Zeilen aus Hexagonen. Dabei werden die Zeilen mit einem geraden Index von jeweils einem Kernel mit einem geraden Index berechnet, während die Zeilen mit einem ungeraden Index jeweils von einem Kernel mit einem ungeraden Index berechnet werden. Zusätzlich gilt, dass sich beim hexagonalen Tiling senkrecht zu der Systemdimension Spalten aus Hexagonen bilden. Dabei werden die ungeraden Spalten von sämtlichen Kernels mit einem ungeraden Index berechnet, während die Spalten mit einem geraden Index von sämtlichen Kernels mit einem geraden Index berechnet werden.

Um die Tile-Höhen und die Tile-Breiten für ein hexagonales Tiling zu wählen, existieren folgende Freiheitsgrade und Zwangsbedingungen: Alle Hexagone müssen gleich hoch sein. Des Weiteren müssen alle Hexagone in den ungeraden Spalten jeweils gleich breit sein und alle Hexagone in den geraden Spalten müssen ebenfalls jeweils gleich breit sein. Zudem muss für die Breite w und für die Höhe h eines Hexagons folgende Ungleichung erfüllt sein:

$$w - 2d\left(\frac{h}{2} - 1\right) \geq d \tag{10.2}$$

Als zusätzlicher Freiheitsgrad können wir die Hexagone noch entlang der Abhängigkeitskette um eine Anzahl von s RHS $\rightarrow$ LC-Gliedern nach unten verschieben, wobei diese Verschiebung zwischen 0 und $h/2$ Glieder betragen kann. Durch diese Verschiebung hören die Hexagone des Kernels k_1 dann nicht mehr im Glied $h/2$, sondern im Glied $h/2 - s$ auf, während die Hexagone des Kernels k_3 im Glied $h/2 - s + 1$ anfangen und die Hexagone des Kernels k_2 im Glied $h - s$ aufhören. Analog versetzt diese Verschiebung auch die Hexagone aller anderen Kernels dementsprechend ebenfalls um s Glieder nach unten.

10.2.8 Datenwiederverwendung

Die Grundkonzepte der Datenwiederverwendung in einem Tile sind analog zu den Grundkonzepten der Datenwiederverwendung in einem Block bei einer Variante mit Kernelfusion:

- Datentransfer für fusionierte (RHS | LC | MAP) $\rightarrow$ (LC | MAP | RED)-Abhängigkeiten sowie für LC $\rightarrow$ RHS-Abhängigkeiten über den On-Chip-Speicher des Prozessors,
- Datenwiederverwendung für permanente Vektoren aus dem DRAM, die von mehreren fusionierten Vektorgrundoperationen des Kernels als Argument gelesen werden, sowie

- kein unnötiges Zurückschreiben von Ergebnissen von (RHS | LC | MAP)-Operationen von dem On-Chip-Speicher in den permanenten Vektor und damit in den DRAM.

Das Tiling behandelt diese drei Fälle, also ob ein Tile die benötigten Komponenten eines Vektors selbst produziert oder aus dem dazugehörigen permanenten Vektor und damit aus DRAM laden muss, und ob ein Tile selbst produzierte Komponenten eines Vektors in den dazugehörigen permanenten Vektor und damit in den DRAM zurückschreiben muss, indem es dafür für jeden Vektor entsprechende Intervalle für die aus dem DRAM zu lesenden Komponenten (*DRAM-Leseintervalle*) und für die in den DRAM zu schreibenden Komponenten (*DRAM-Schreibintervalle*) definiert.

Für diejenigen Komponenten, die ein Tile von einem benötigten Vektor selbst produziert und wiederverwenden muss oder die ein Tile aus dem permanenten Vektor und damit aus dem DRAM laden muss, unterscheidet das Tiling zwischen folgenden Fällen (siehe Abbildung 10.5):

- das Tile produziert alle benötigten Komponenten des Vektors selbst und muss keine der benötigten Komponenten dieses Vektors aus dem DRAM laden,
- das Tile muss nur die benötigten Komponenten des Vektors am rechten und linken Rand des Tiles aus dem DRAM laden, während es die benötigten Komponenten in der Mitte des Tiles selbst erzeugt, oder
- das Tile muss alle benötigten Komponenten des Vektors aus dem DRAM laden.

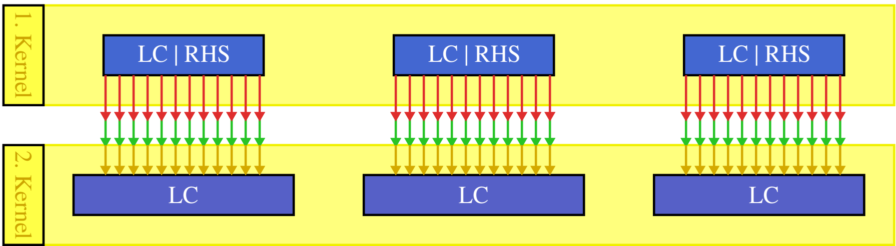
Ebenso muss das Tiling für diejenigen Komponenten, die ein Tile von einem Vektor selbst produziert hat und in den permanenten Vektor beziehungsweise in den DRAM zurückschreiben muss, folgende Fälle behandeln (siehe Abbildung 10.5):

- das Tile muss keine selbst produzierten Komponenten des Vektors in den DRAM zurückschrieben,
- das Tile muss nur die selbst produzierten Komponenten des Vektors am rechten und linken Rand des Tiles in den DRAM zurückschrieben,
- das Tile muss alle selbst produzierten Komponenten des Vektors in den DRAM zurückschreiben.

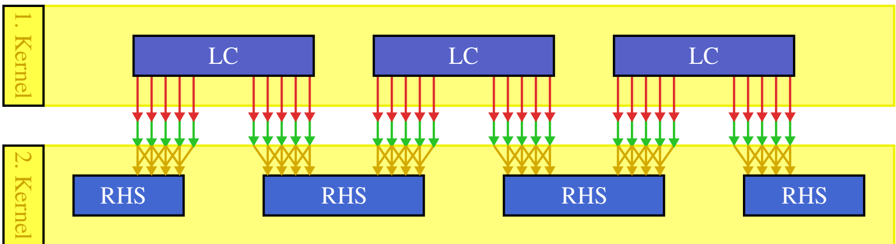
10.2.9 Tiling über die Stufen eines ODE-Verfahrens

Falls ein Einschrittverfahren eine Schrittweitensteuerung besitzt, dann wirkt diese wie eine globale Barriere am Ende eines jeden Zeitschritts. Dadurch können wir bei einem Einschrittverfahren mit Schrittweitensteuerung nur das *Tiling über die Stufen* des Verfahrens durchführen, das heißt das Tiling muss mit dem ersten Glied der RHS $\rightarrow$ LC-Abhängigkeitskette beginnen und mit dem letzten Glied der RHS $\rightarrow$ LC-Abhängigkeitskette aufhören. Es ist jedoch bei einem ODE-Verfahren mit Schrittweitensteuerung unmöglich, dass ein Tile über die Zeitschrittgrenzen hinweg verläuft. Deshalb ist es bei solchen Verfahren ebenfalls unmöglich, Daten über die Zeitschrittgrenzen wiederzuverwenden.

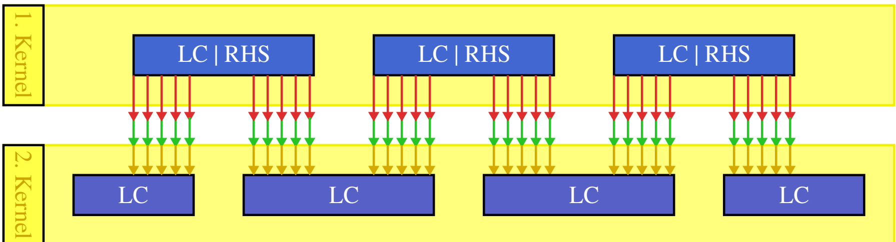
Systemdimension



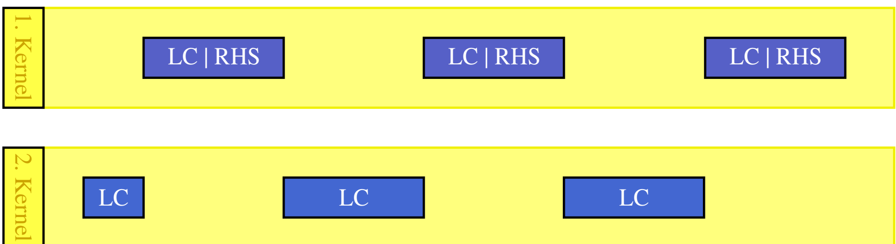
Alle Komponenten werden für eine (LC | RHS)→LC-Abhängigkeit übertragen



Die rechten und linken Kompo. werden für eine LC→RHS-Abhängigkeit übertragen



Die rechten und linken Kompo. werden für eine (LC | RHS)→LC-Abhängigkeit übertragen



Keine Komponenten werden für eine (LC | RHS)→LC-Abhängigkeit übertragen

Abbildung 10.5: Zwischen zwei Kernels über den DRAM transferierte Vektorkomponenten.

10.2.10 Tiling über die Zeitschritte eines ODE-Verfahrens

10.2.10.1 Grundidee

Falls ein ODE-Verfahren keine Schrittweitensteuerung besitzt, so existiert auch keine globale Barriere am Ende eines jeden Zeitschritts mehr. Somit wird für solche ODE-Verfahren die RHS $\rightarrow$ LC-Abhängigkeitskette zu einem Abhängigkeitszyklus. Folglich ist das Tiling nicht nur über die Stufen eines einzigen Zeitschritts erlaubt, sondern es ist auch ein *Tiling über die Zeitschritte* möglich (siehe Abbildung 10.6). Für dieses Tiling über die Zeitschritte kann nun jedes Glied der Abhängigkeitskette als Startpunkt und Endpunkt für das Tiling dienen. Dadurch kann ein Tile Vektorgrundoperationen aus mehreren unterschiedlichen Zeitschritten berechnen, weshalb beim Tiling über die Zeitschritte auch kein glatter Schnitt mehr über die Tiles an den Zeitschrittgrenzen existiert. Auf diese Weise ermöglicht das Tiling auch eine Datenwiederverwendung über die Zeitschrittgrenzen hinweg.

Dieses Tiling über die Zeitschritte ist gerade bei Einschrittverfahren mit niedriger Ordnung oder bei Mehrschrittverfahren mit hoher Ordnung vielversprechend, da diese Verfahren einen großen Anteil von ihren Vektoren von einem Zeitschritt in den nächsten Zeitschritt transferieren müssen. Dahingegen ist das Tiling über die Zeitschritte bei Einschrittverfahren mit hoher Ordnung weniger vielversprechend, da diese zwar in einem Zeitschritt über ihre vielen Stufen viele Vektoren erzeugen, aber nur einen einzigen Vektor von einem Zeitschritt in den nächsten Zeitschritt transferieren müssen.

Für das Tiling über die Zeitschritte müssen wir folgende Erweiterungen einführen:

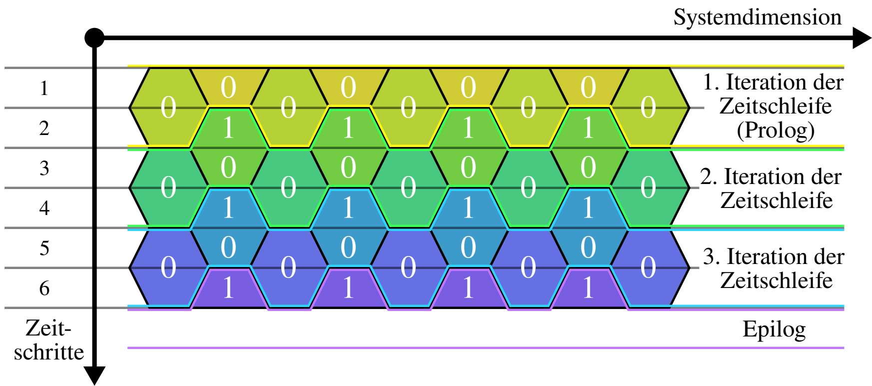
1. eine *Abroll-Transformation* des Datenflussgraphen, um sich wiederholende Tiling-Muster als Grundlage für die Zeitschleife zu erzeugen, bei der nun eine Iteration mehrere Zeitschritte umspannt,
2. eine *Zeit-Mapping-Distanz* zwischen Kernels und Vektorgrundoperationen, um Tiles zu modellieren, die sich nicht komplett innerhalb einer Iteration der Zeitschleife befinden, sondern sich mit zwei Iterationen der Zeitschleife überlappen, sowie
3. die Erzeugung einer speziellen *Prologiteration* und einer *Epilogiteration* für die Zeitschleife.

10.2.10.2 Abrollen des Graphens

10.2.10.2.1 Abroll-Transformation

Um im allgemeinen Fall das Tiling über die Zeitschritte zu verwirklichen, müssen wir den Datenflussgraph über eine *Abroll-Transformation* abrollen, bevor wir das Tiling anwenden. Da diese Abroll-Transformation an sich die Performance nicht direkt, sondern nur in Kombination mit dem Tiling erhöht, lässt sie sich als eine weitere Enabling-Transformation betrachten. Für diese Abroll-Transformation definieren wir einen *Abrollfaktor* u, so dass nach dem Abrollen des Datenflussgraphen eine Iteration der Zeitschleife nun u Zeitschritten entspricht. Dabei ist die Wahl des Abrollfaktors u von den Tile-Höhen des gewählten Tiling-Schemas abhängig.

Um den abgerollten Graph zu generieren, erzeugen wir zunächst einen leeren Datenflussgraphen, der später der abgerollte Graph von dem ursprünglichen Graph werden soll. Dann fügen



Legende:

- 1. Iteration des 1. Kernels (Prolog)
- 1. Iteration des 2. Kernels (Prolog)
- 2. Iteration des 1. Kernels
- 2. Iteration des 2. Kernels
- 3. Iteration des 1. Kernels
- 3. Iteration des 2. Kernels
- 4. Iteration des 1. Kernels (Epilog)

Abbildung 10.6: Tiling über die Zeitschritte. Diese Abbildung zeigt das Tiling über die Zeitschritte für ein hexagonales Tiling. In ihr besitzt ein Hexagon eine Höhe von zwei Zeitschritten, wofür der Datenflussgraph auch zweimal abgerollt wurde. Die Zahlen in den Hexagonen geben zusätzlich die Zeit-Mapping-Distanz der Vektorgrundoperationen in den Kernels an.

wir u Kopien von allen Grundoperationen des ursprünglichen Graphen in den neuen Graphen ein, und weisen eine *Abroll-ID* u_{id} angefangen von 0 bis $u-1$ den u Kopien zu. Als Nächstes müssen wir für jede Abhängigkeit $d : a \rightarrow b$ im ursprünglichen Graphen u Abhängigkeiten $d_{\text{unrolled}} : a_{\text{unrolled}} \rightarrow b_{\text{unrolled}}$ mit den Abroll-IDs von 0 bis $u-1$ in den abgerollten Graphen einfügen. Um die Abroll-ID des Ursprungs a_{unrolled} und die Abroll-ID des Ziels b_{unrolled} und die Zeitschrittdistanz $\Delta\kappa_u$ für eine abgerollte Abhängigkeit d_{unrolled} zu bestimmen, müssen wir die folgenden Regeln verwenden:

$$\Delta\kappa_u(d_{\text{unrolled}}) = \left\lfloor \frac{\Delta\kappa(d) + u_{\text{id}}(d_{\text{unrolled}})}{u} \right\rfloor \tag{10.3}$$

$$u_{\text{id}}(a_{\text{unrolled}}) = u_{\text{id}}(d_{\text{unrolled}}) \tag{10.4}$$

$$u_{\text{id}}(b_{\text{unrolled}}) = \Big(\Delta\kappa(d) + u_{\text{id}}(d_{\text{unrolled}})\Big) \bmod u \tag{10.5}$$

10.2.10.2.2 Bestimmung des Abrollfaktors für das hexagonale Tiling

Wie wir bereits zuvor erwähnten, so müssen beim hexagonalen Tiling-Schema alle Hexagone die gleiche Höhe h besitzen. Damit wir beim Tiling über die Zeitschritte das hexagonale

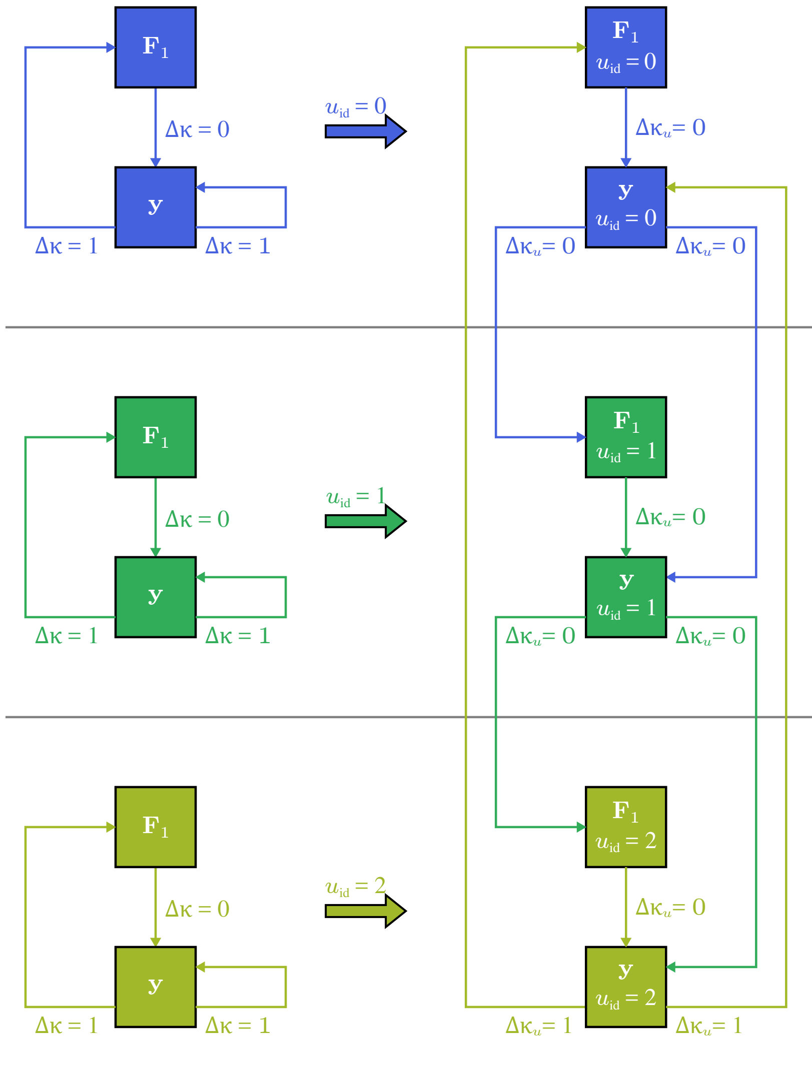


Abbildung 10.7: Beispiel für die Anwendung der Abrolltransformation auf das explizite Eulerverfahren.

Tiling-Schema ohne Unterbrechung fortsetzen können, muss deshalb die Länge des abgerollten Abhängigkeitszyklus das kleinste gemeinsame Vielfache von der Höhe der Hexagone h und der ursprünglichen Länge des Abhängigkeitszyklus l sein. Dies resultiert in einem Abrollfaktor u von:

$$u = \frac{\text{kgV}(h, l)}{l} \tag{10.6}$$

Die Abbildung 10.6 illustriert eine Zeitschleife mit einem ununterbrochenen hexagonalen Tiling-Schema mit einem Abrollfaktor von $u = 2$. Wenn wir jedoch für eine bestimmte Hexagon-Höhe einen anderen Abrollfaktor als in der Gleichung 10.6 wählen, so müssen wir das hexagonale Tiling-Schema in einem Glied des RHS $\rightarrow$ LC-Abhängigkeitszyklus unterbrechen und im nächsten Glied reinitialisieren. Dieses Unterbrechen und Reinitialisieren erfordert, dass wir in dem entsprechenden RHS $\rightarrow$ LC-Glied die schrumpfenden und wachsenden Hexagone abschneiden, beziehungsweise die dazugehörigen Kernels abschließen, und im nächsten RHS $\rightarrow$ LC-Glied das hexagonale Tiling-Schema über zwei neue Kernels, eins für die wachsenden und eins für die schrumpfenden Hexagone, reinitialisieren. Durch dieses Abschneiden und Reinitialisieren zwischen zwei aufeinanderfolgenden RHS $\rightarrow$ LC-Gliedern ist offensichtlich keine Datenwiederverwendung über diese RHS $\rightarrow$ LC-Glieder hinweg möglich.

10.2.10.2.3 Bestimmung des Abrollfaktors für das trapezoidale Tiling

Im Gegensatz zum hexagonalen Tiling-Schema können beim trapezoidalen Tiling-Schema die Zeilen aus Trapezoiden jeweils unterschiedliche Höhen besitzen. Diese Höhen müssen wir beim trapezoidalen Tiling-Schema so wählen, dass deren Summe ein Vielfaches der Länge der Abhängigkeitskette ist. Dieses Vielfache entspricht dann dem Abrollfaktor u. Für denjenigen Sonderfall, dass die Reihen aus Trapezoiden alle die gleiche Höhe besitzen sollen, kann Formel wie beim hexagonalen Tiling-Schema (Gleichung 10.6) für die Bestimmung des Abrollfaktors verwendet werden.

10.2.10.3 Zeit-Mapping-Distanz

10.2.10.3.1 Grundidee

Da beim Tiling über die Zeitschritte keine glatten Schnitte entlang der Zeitschrittgrenzen zwischen den Tiles mehr existieren, ist es nötig zu definieren, ob ein Tile beziehungsweise ein Kernel für eine Vektorgrundoperation Komponenten von der aktuellen oder der vorherigen Iteration der Zeitschleife berechnet.

Für diesen Zweck definieren wir eine *Zeit-Mapping-Distanz* $\Delta m(k, o) \in \{0, 1\}$ für das Mapping einer Vektorgrundoperation o auf ein Kernel k, wie folgt (siehe Abbildung 10.6 für eine Illustration): Angenommen, dass bei einem Abrollfaktor u die Iteration $i = 1, 2, \ldots$ der Zeitschleife mit den Zeitschritten $\{(i-1) \cdot u+1, \ldots, i \cdot u\}$ assoziiert wird, dann impliziert eine Zeit-Mapping-Distanz von 0, dass das Kernel die Vektorgrundoperation o für einen Zeitschritt κ aus der gegenwärtigen Iteration der Zeitschleife berechnet, das heißt $\kappa \in \{(i-1) \cdot u+1, \ldots, i \cdot u\}$. Falls die Zeit-Mapping-Distanz 1 ist, dann berechnet das Kernel die Vektorgrundoperation o für einen Zeitschritt κ aus der vorherigen Iteration der Zeitschleife, das heißt $\kappa < (i-1) \cdot u + 1$.

10.2.10.3.2 Abhängigkeiten zwischen zwei Kernels

Beim Tiling über die Zeitschritte existiert eine Abhängigkeit zwischen zwei Kernels $k_1 \to k_2$ nur, falls für mindestens eine Abhängigkeit zwischen zwei Vektorgrundoperationen $a \to b$, wobei a mit k_1 und b mit k_2 fusioniert wird, sich die Eingabe- und Ausgabeintervalle für mindestens ein Paar von Blöcken i und j beziehungsweise ein Paar von Tiles überlappen. Für diese beiden Blöcke gilt somit $\mathcal{O}(a, k_1, i) \cap \mathcal{I}(b, k_2, j) \neq \varnothing$. In Abhängigkeit von der Zeit-Mapping-Distanz können wir die folgenden Regeln für die Art der Abhängigkeit zwischen diesen beiden Kernels bezüglich der Zeitschleife definieren:

- **Schleifenunabhängige echte Abhängigkeit:** Eine *schleifenunabhängige echte Abhängigkeit*, das heißt eine Abhängigkeit innerhalb einer Iteration der Zeitschleife, zwischen zwei Kernels tritt auf, falls $\Delta m(k_1, a) - \Delta\kappa_u(a \to b) - \Delta m(k_2, b) = 0$ und zudem $k_1 \neq k_2$ gilt. Denn würde zusätzlich $k_1 = k_2$ gelten, so wäre dies keine Abhängigkeit zwischen zwei Kernels, sondern eine Abhängigkeit zwischen den Vektorgrundoperationen innerhalb eines Kernels.
- **Schleifengetragene echte Abhängigkeit:** Eine *schleifengetragene echte Abhängigkeit*, das heißt eine Abhängigkeit, die über x Iterationen der Zeitschleife spannt, tritt auf, falls $\Delta m(k_1, a) - \Delta\kappa_u(a \to b) - \Delta m(k_2, b) = x < 0$ gilt. Falls zudem $k_1 = k_2$ ist, dann handelt es sich nicht um eine Abhängigkeit innerhalb eines Kernels, sondern um eine Abhängigkeit zwischen zwei unterschiedlichen Ausführungen dieses Kernels in jeweils einer unterschiedlichen Iteration der Zeitschleife.
- **Schleifengetragene Gegenabhängigkeit:** Eine *schleifengetragene Gegenabhängigkeit*, also eine Abhängigkeit von einer zukünftigen Iteration der Zeitschleife zu der gegenwärtigen Iteration, tritt auf, falls $\Delta m(k_1, a) - \Delta\kappa_u(a \to b) - \Delta m(k_2, b) > 0$ gilt. Da unser Ansatz die Iterationen der Zeitschleife nacheinander ausführt, ist eine solche schleifengetragene Gegenabhängigkeit bei unserem Ansatz **verboten**.

Dabei ist zu beachten, dass wir sowohl die schleifenunabhängigen echten Abhängigkeiten als auch die schleifengetragenen echten Abhängigkeiten für den Datentransfer zwischen den Kernels berücksichtigen müssen, während wir nur die schleifenunabhängigen echten Abhängigkeiten für die topologische Sortierung der Kernels beziehungsweise der ausführbaren Objekte in der Zeitschleife berücksichtigen müssen. Eine allgemeine Diskussion über die Arten von Abhängigkeiten in Schleifen kann zum Beispiel in dem Buch [B3] gefunden werden.

10.2.10.3.3 Abhängigkeiten innerhalb eines Kernels

Ein Tile eines Kernels führt beim Tiling über die Zeitschritte weiterhin seine zugewiesenen Vektorgrundoperationen gemäß der topologischen Ordnung ihrer Abhängigkeiten aus. Eine Abhängigkeit $a \to b$ zwischen zwei Vektorgrundoperationen a und b in dem Tile i eines Kernel k existiert nur, falls die folgenden Bedingungen erfüllt sind:

- $\Delta m(k, a) - \Delta\kappa(a \to b) - \Delta m(k, b) = 0$
- $\mathcal{O}(a, k, i) \cap \mathcal{I}(b, k, i) \neq \varnothing$

10.2.10.4 Prolog- und Epilogiteration

Da beim Tiling über Zeitschritte kein glatter Schnitt mehr über die Tiles an den Zeitschrittgrenzen existiert, benötigen wir eine besondere *Prologiteration* und eine *Epilogiteration* (siehe Abbildung 10.6). Diese beiden bestehen jeweils aus spezialisierten Prolog- und Epilog-Kernels, von denen jedes jeweils einem regulären Kernel entspricht. Dabei führt die Zeitschrittprozedur die Prologiteration anstelle der ersten Iteration der Zeitschleife aus. In dieser Prologieration ruft die Zeitschrittprozedur nur die spezialisierten Prologkernels auf. Dabei berechnen die Prologkernels nur diejenigen Vektorgrundoperationen ihres regulären Kernels, für welche das reguläre Kernel eine Zeit-Mapping-Distanz $\Delta m = 0$ besitzt. Dann führt die Zeitschrittprozedur die gewünschte Anzahl an regulären Iterationen der Zeitschleife aus. Hierbei ruft die Zeitschrittprozedur für jede reguläre Iteration nur die regulären Kernels auf. Nach der letzten regulären Iteration der Zeitschleife führt die Zeitschrittprozedur die Epilogiteration aus. In dieser Epilogiteration ruft die Zeitschrittprozedur nur die spezialisierten Epilogkernels auf. Dabei berechnen die Epilogkernels nur diejenigen Vektorgrundoperationen ihres regulären Kernels, für welche das reguläre Kernel eine Zeit-Mapping-Distanz $\Delta m = 1$ besitzt.

10.2.11 Autotuning-Ansatz für die Parameter der Tiling-Schemata

10.2.11.1 Grundlegendes

Je höher und je breiter die Tiles einer Implementierungsvariante mit Tiling sind, umso geringer ist auch das Datenvolumen der Implementierungsvariante. Allerdings besitzen höhere und breitere Tiles auch einen größeren Workingset. Dadurch reduziert eine Erhöhung der Tile-Höhe und der Tile-Breite, sobald der Workingset aller nebenläufig ausgeführten Tiles die Größe des On-Chip-Speichers des Prozessors übersteigt, auch die Cache-Effizienz der Implementierungsvariante, wodurch sich wiederum die Datenwiederverwendung über den On-Chip-Speicher verringert und das DRAM-Volumen erhöht. Da sich der Performance-Gewinn der Implementierungsvarianten mit Tiling gegenüber den Implementierungsvarianten mit Fusion jedoch primär aus dem reduzierten DRAM-Volumen ergibt, müssen wir die Tile-Breite, die Tile-Höhe und die Anzahl an Kerne, welche an einem Tile zusammenarbeiten, geschickt so wählen, dass wir zwar ein kleines Datenvolumen erzielen, jedoch zusätzlich der Workingset der nebenläufig ausgeführten Tiles die Größe des On-Chip-Speichers des Prozessors nicht übersteigt. Bei den Implementierungsvarianten mit Tiling sind im Gegensatz zu den Implementierungsvarianten mit Fusion allerdings das Datenvolumen und das DRAM-Volumen, wie die Experimente im Abschnitt 10.3.2 und 10.4.2 zeigen werden, auf Grund von Latenzen und anderen Flaschenhälsen nicht mehr in etwa direkt proportional zur Performance.

Sowohl das trapezoidale als auch das hexagonale Tiling-Schema besitzen für ihre Parameter viele Freiheitsgrade, welche leider eine kombinatorische Explosion verursachen. Hierdurch müsste zum Beispiel ein Autotuning-Ansatz für die Parameter des trapezoidalen Tilings, welcher auf einer vollständigen Suche basiert, bei 5 Zeilen aus Trapezoiden und 10 vielversprechenden Breiten für jede Zeile bereits die Performance für circa $10^5 \cdot 10^5$ mögliche Parameterkombinationen evaluieren. Somit muss unser Autotuning-Ansatz die Bestimmung der besten Parameter des Tilings in Teilprobleme zerlegen, die er getrennt und unabhängig voneinander optimieren kann.

Erschwerend kommt hinzu, dass jedes Kernel eine plattformspezifische *Kernelkonfiguration* besitzt, welche das Autotuning ebenfalls optimieren sollte. Diese Kernelkonfiguration um-

fasst je nach Zielarchitektur unter anderem bei GPUs die Anzahl an Workgroups pro Tile, die Workgroup-Größe, die Strides und die Blöcke pro Thread, sowie bei CPUs die Anzahl an Threads pro Tile und die Schleifenstruktur mit ihren Blockgrößen. In dieser Arbeit werden wir das Autotuning abstrakt gestalten, ohne dabei auf die spezifischen Eigenschaften der jeweiligen Plattform beziehungsweise Zielarchitektur einzugehen. Deshalb bestimmen wir in dieser Arbeit die beste Kernelkonfiguration für jedes Kernel immer durch eine vollständige Suche über eine Menge von vielversprechenden Kernelkonfigurationen. Aus diesem Grund müssen wir jedoch auch die Menge an vielversprechenden Kernelkonfigurationen entsprechend klein wählen, da die vollständige Suche ansonsten zu lange dauern würde. Zusätzlich evaluieren wir in dieser Arbeit die Performance eines Kernels mit Tiling nicht über ein Performance-Modell, sondern nur über ein kostenintensives Mikrobenchmark, welches das Kernel kompiliert und ausführt. Deshalb sollte unser Ansatz zum Autotuning für das Tiling es vermeiden, für ein und dasselbe Kernel das Mikrobenchmark redunanterweise mehrmals auszuführen.

Im Folgenden stellen wir exemplarisch einen Autotuning-Ansatz vor, welcher nur die Parameter für ein Tiling über die Stufen, nicht aber die Parameter für ein zusätzliches Tiling über die Zeitschritte optimiert. Allerdings lässt sich der vorgestellte Ansatz problemlos durch zusätzliche Sonderfälle und deren Behandlung auch auf ein Tiling über die Zeitschritte erweitern.

10.2.11.2 Trapezoidales Tiling-Schema

Das trapezoidale Tiling-Schema besitzt folgende Parameter, welche wir geschickt wählen müssen:

- eine Höhe $h_{i\text{te Zeile}}$ für jede Zeile i aus Trapezoiden,
- eine Breite $w_{i\text{te Zeile, gerade Trapezoide}}$ und eine plattformspezifische Kernelkonfiguration $c_{i\text{te Zeile, gerade Trapezoide}}$ für die geraden Trapezoide einer Zeile i, sowie
- eine Breite $w_{i\text{te Zeile, ungerade Trapezoide}}$ und eine plattformspezifische Kernelkonfiguration $c_{i\text{te Zeile, ungerade Trapezoide}}$ für die ungeraden Trapezoide einer Zeile i.

Dabei ist eine Zeile i an Trapezoiden eindeutig über ihr Startglied s_i in der Abhängigkeitskette, ihre Höhe h_i sowie die Breite $w_{i\text{te Zeile, gerade Trapezoide}}$ der geraden Trapezoiden und die Breite $w_{i\text{te Zeile, ungerade Trapezoide}}$ der ungeraden Trapezoiden bestimmt. Bei einem vollständigen trapezoidalen Tiling-Schema bestehend aus z Zeilen aus Trapezoiden muss die Summe über sämtliche Höhen zudem die Länge der Abhängigkeitskette l ergeben:

$$l = \sum_{i=1}^{z} h_{i\text{te Zeile}}. \tag{10.7}$$

Für das Startglied s_i einer Zeile i gilt somit:

$$s_{i\text{te Zeile}} = \sum_{j=1}^{i-1} h_{j\text{te Zeile}}. \tag{10.8}$$

Um einen Ansatz zur Bestimmung der optimalen Parameter für das trapezoidale Tiling-Schema zu entwickeln, stellen wir zunächst einige theoretische Überlegungen an. Als Erstes nehmen

wir an, dass die Gesamtlaufzeit T_{gesamt} eines Zeitschritts die Summe der Laufzeit sämtlicher Zeilen sei:

$$T_{\text{gesamt}} = \sum_{i=1}^{z} T_{i\text{te Zeile}}. \tag{10.9}$$

Die Laufzeit $T_{i\text{te Zeile}}$ einer Zeile i setzt sich zusammen aus der Laufzeit $T_{i\text{te Zeile, gerade Trapezoide}}$ des Kernels für die geraden Trapezoide und aus der Laufzeit $T_{i\text{te Zeile, ungerade Trapezoide}}$ des Kernels für die ungeraden Trapezoide:

$$T_{i\text{te Zeile}} = T_{i\text{te Zeile, gerade Trapezoide}} + T_{i\text{te Zeile, ungerade Trapezoide}}. \tag{10.10}$$

Die Laufzeiten beider Kernels einer Zeile i auf einem Prozessor wiederum lassen sich über dessen Durchsatz $D_{i\text{te Zeile, gerade Trapezoide}}$ an geraden Trapezoiden und dessen Durchsatz $D_{i\text{te Zeile, ungerade Trapezoide}}$ an ungeraden Trapezoiden abschätzen:

$$\begin{aligned} T_{i\text{te Zeile, gerade Trapezoide}} &= \frac{N_{i\text{te Zeile}}}{D_{i\text{te Zeile, gerade Trapezoide}}}, \\ T_{i\text{te Zeile, ungerade Trapezoide}} &= \frac{N_{i\text{te Zeile}}}{D_{i\text{te Zeile, ungerade Trapezoide}}}. \end{aligned} \tag{10.11}$$

Dabei ist $N_{i\text{te Zeile}}$ die Anzahl der geraden beziehungsweise ungeraden Trapezoiden in der Zeile i, die sich aus der Systemgröße n und der beschränkten Zugriffsdistanz d wie folgt errechnet:

$$N_{i\text{te Zeile}} = \frac{n}{w_{i\text{te Zeile, gerade Trapezoide}} + w_{i\text{te Zeile, ungerade Trapezoide}} - (h_{i\text{te Zeile}} - 1) \cdot d}. \tag{10.12}$$

Hier nehmen wir zur Vereinfachung an, dass der Durchsatz $D_{i\text{te Zeile, gerade Trapezoide}}$ an geraden Trapezoiden einer Zeile i nur von der Höhe h_i der Zeile, von der Kernelkonfiguration $c_{i\text{te Zeile, gerade Trapezoide}}$ der geraden Trapezoide und von der Breite $w_{i\text{te Zeile, gerade Trapezoide}}$ der geraden Trapezoide abhängig ist, nicht aber von dem Stride zwischen den Trapezoiden, welcher zusätzlich von der Breite $w_{i\text{te Zeile, ungerade Trapezoide}}$ der ungeraden Trapezoide beeinflusst wird. Analoges nehmen wir für die ungeraden Trapezoide an.

Diese theoretischen Überlegungen können wir nun in einen Ansatz zur Bestimmung der besten Parameter für das trapezoidale Tiling-Schema umsetzen. Dieser Ansatz gliedert sich in zwei Phasen:

- **1. Phase: Suche nach den besten Parametern und der besten Laufzeit für jede mögliche Zeile.** Für jede mögliche Zeile, also jedes mögliche Paar von Zeilenstartglied s und Zeilenhöhe h, bestimmen wir in diesem Schritt jeweils die besten Breiten und Kernelkonfigurationen, für welche die Laufzeit dieser Zeile minimal ist. Dies können wir über zwei unterschiedliche Ansätze erreichen:
  - **1. Ansatz zur Bestimmung der Breiten: Zweidimensionale Suche.** Hierbei nutzen wir aus, dass die beiden Breiten einer Zeile einen zweidimensionalen und relativ glatten Konfigurationsraum aufspannen, wodurch wir in diesem Konfigurationsraum mit einem entsprechenden Suchalgorithmus, wie zum Beispiel Simulated Annealing oder Nelder-Mead, eine performante Konfiguration finden können. Dabei

muss diese Suche zusätzlich für jede Konfiguration in diesem zweidimensionalen Konfigurationsraum, die der verwendete Suchalgorithmus abtasten möchte, die optimale Kernelkonfiguration von dem Kernel der geraden Trapezoide und von dem Kernel der ungeraden Trapezoide bestimmen. Dies kann wiederum, ohne auf die Eigenschaften der Zielarchitektur einzugehen, über eine vollständige Suche über eine vorgegebene Menge an vielversprechenden Kernelkonfigurationen geschehen, wobei die Suche die beiden Kernelkonfigurationen auf Grund von Gleichung 10.10 unabhängig voneinander optimieren kann. Somit muss diese Suche über die beiden Breiten jeweils für jede abzutastende Konfiguration im zweidimensionalen Konfigurationsraum und zusätzlich für jede Kernelkonfiguration zwei Kernel für den Zielrechner kompilieren und auf dem Zielrechner ausführen.

  – **2. Ansatz zur Bestimmung der Breiten: Benchmarking einer Menge von vielversprechenden Breiten und Kombination der Benchmark-Ergebnisse.** Hier nutzen wir aus, dass der Durchsatz eines Prozessors an geraden Trapezoiden in einer Zeile nur von deren Breite und deren Kernelkonfiguration nicht aber von der Breite der ungeraden Trapezoide abhängig ist, und vice versa. Somit können wir für die geraden Trapezoide einer Zeile für eine Menge von gegebenen Breiten und einer Menge von vielversprechenden Kernelkonfigurationen über eine vollständige Suche für jede Breite jeweils die optimale Kernelkonfiguration und damit den höchsten Durchsatz (siehe Gleichungen 10.11 und 10.12) bestimmen. Analog können wir für die ungeraden Trapezoide einer Zeile vorgehen. Somit müssen wir für die Bestimmung dieser Durchsätze sowohl für die ungeraden als auch für die geraden Trapezoide für jede Breite und für jede Kernelkonfiguration jeweils ein Kernel kompilieren und ausführen. Anschließend können wir ohne weiteres Benchmarking nur über die Kombination der gemessenen Durchsätze durch die Gleichungen 10.10, 10.11 und 10.12 diejenigen Breiten für die geraden und ungeraden Trapezoide bestimmen, für welche die Laufzeit der Zeile minimal ist.

Hierbei gilt anzumerken, dass auf CPUs die Tile-Breite zwar ein Vielfaches der Cache-Line-Größe sein sollte, ansonsten aber frei gewählt werden kann, wodurch sich auf CPUs wiederum sehr viele vielversprechende Tile-Breiten mit demselben Abstand zueinander ergeben. Deswegen scheint für CPUs zur Bestimmung der besten Parameterkombination einer Zeile die soeben genannte zweidimensionale Suche besser geeignet zu sein. Dahingegen gibt es auf einer GPU auf Grund der Occupancy-Regeln nur wenige vielversprechende Tile-Breiten, welche zusätzlich zueinander noch einen unregelmäßigen Abstand besitzen. Deswegen scheint es auf GPUs vielversprechender zu sein, die beste Parameterkombination einer Zeile über das Benchmarking dieser Menge an vielversprechenden Tile-Breiten mit einer anschließenden Kombination der Benchmark-Ergebnisse zu finden.

- **2. Phase: Kombination der Zeilen.** Ist die beste Parameterkonfiguration und die beste Laufzeit für jede mögliche Zeile gefunden, dann können wir in diesem Schritt ohne weiteres Benchmarking diejenige Kombination an Zeilen beziehungsweise Zeilenhöhen finden, die sich über die gesamte Abhängigkeitskette erstreckt und deren aufsummierte Laufzeiten minimal sind, wodurch wir nach Gleichung 10.9 auch die Gesamtlaufzeit eines Zeitschritts minimieren. Die Suche nach sämtlichen möglichen Kombinationen von Zeilenhöhen lässt sich auf die Suche nach sämtlichen möglichen Zerlegungen der Länge der Abhängigkeitskette in ihre Summanden zurückführen, wobei jeder Summand die Hö-

he einer Zeile aus Trapezoiden repräsentiert und die Zerlegungen sich über die Reihenfolge der Summanden unterscheiden. So lässt sich beispielsweise eine Abhängigkeitskette der Länge 4 in folgende Zeilenhöhen zerlegen: 4, 1+3, 3+1, 2+2, 2+1+1, 1+2+1, 1+1+2, 1+1+1+1. Eine solche Zerlegung können wir leicht rekursiv implementieren.

Über diese zwei Phasen erreicht unser Autotuning-Ansatz, dass er nicht für ein und dasselbe Kernel, das heißt ein Kernel mit einem bestimmten Startglied, einer bestimmten Höhe, einer bestimmten Breite und einer bestimmten Kernelkonfiguration mehrmals redundanterweise durch ein kostenintensives Benchmark dessen Laufzeit beziehungsweise dessen Durchsatz ermitteln muss. Man könnte diesen Ansatz noch weiter verfeinern, so dass er noch weniger Kernels einem Benchmark unterziehen muss. Dieses Verfeinern könnten wir zum Beispiel durch ein Ausästen des Suchraums über Performance-Modelle realisieren.

10.2.11.3 Hexagonales Tiling-Schema

Das hexagonale Tiling-Schema besitzt folgende Parameter, welche wir geschickt wählen müssen:

- eine gemeinsame Höhe h für alle Hexagone,
- eine Verschiebung s der Hexagone entlang der Abhängigkeitskette,
- eine Breite $w_{\text{gerade Zeilen}}$ für die geraden Zeilen aus Hexagonen,
- eine Breite $w_{\text{ungerade Zeilen}}$ für die ungeraden Zeilen aus Hexagonen,
- eine plattformspezifische Kernelkonfiguration $c_{i\text{te gerade Zeile}}$ für jede gerade Zeile i aus Hexagonen, sowie
- eine plattformspezifische Kernelkonfiguration $c_{j\text{te ungerade Zeile}}$ für jede ungerade Zeile j aus Hexagonen.

Insgesamt besteht ein hexagonales Tiling entlang einer Abhängigkeitskette der Länge l aus $\lfloor l/h \rfloor$ geraden Zeilen aus Hexagonen und aus $\lfloor l/h \rfloor + 1$ ungeraden Zeilen aus Hexagonen.

Somit besitzt das hexagonale Tiling-Schema ähnliche Parameter wie das trapezoidale Tiling-Schema. Im Gegensatz zum trapezoidalen Tiling-Schema existieren beim hexagonalen Tiling-Schema nur ein Freiheitsgrad für die Tile-Höhe und zwei Freiheitsgrade für die Tile-Breiten, wodurch sich beim hexagonalen Tiling-Schema die Bestimmung der besten Parameter stark vereinfacht. Allerdings lohnen sich beim hexagonalen Tiling-Schema meist nur kleine Tile-Höhen (2, 4, 6, und 8), wodurch die Verschiebung entlang der Abhängigkeitskette ebenfalls nur wenige Werte annehmen kann (je nach Höhe 0, 1, 2 oder 3). Da es somit für die Tile-Höhe und die Verschiebung nur wenige Werte gibt, eignen sich beide nicht als Dimensionen für einen mehrdimensionalen Suchalgorithmus. Deshalb scheint beim hexagonalen Tiling-Schema derjenige allgemeine Ansatz, welcher die Tile-Höhe, die Verschiebung entlang der Abhängigkeitskette und die beiden Tile-Breiten als einen vierdimensionalen Suchraum betrachtet und darauf einen entsprechenden Suchalgorithmus anwendet, nicht zielführend zu sein, weshalb wir einen spezialisierten Ansatz entwickeln müssen.

Um einen Ansatz zur Bestimmung der optimalen Parameter des hexagonalen Tilings zu entwickeln, stellen wir zunächst wieder einige theoretische Überlegungen an. Hier nehmen wir zunächst an, dass die Gesamtlaufzeit T_{gesamt} eines Zeitschritts die Summe der gesamten Laufzeit $T_{\text{gerade Zeilen}}$ der geraden Zeilen und der gesamten Laufzeit $T_{\text{ungerade Zeilen}}$ der ungeraden Zeilen sei:

$$T_{\text{gesamt}} = T_{\text{gerade Zeilen}} + T_{\text{ungerade Zeilen}}. \tag{10.13}$$

Dabei setzen sich sowohl die gesamte Laufzeit der geraden Zeilen wiederum aus den Laufzeiten der einzelnen geraden Zeilen beziehungsweise deren Kernels zusammen, während sich die gesamte Laufzeit der ungeraden Zeilen wiederum aus den Laufzeiten der einzelnen ungeraden Zeilen beziehungsweise deren Kernels zusammensetzt:

$$\begin{aligned} T_{\text{gerade Zeilen}} &= \sum_{i=1}^{\lfloor l/h \rfloor} T_{i\text{te gerade Zeile}}, \\ T_{\text{ungerade Zeilen}} &= \sum_{j=1}^{\lfloor l/h \rfloor + 1} T_{j\text{te ungerade Zeile}}. \end{aligned} \tag{10.14}$$

Die Laufzeit eines Kernels einer geraden Zeile i auf einem Prozessor lässt sich über dessen Durchsatz $D_{i\text{te gerade Zeile}}$ an Hexagonen abschätzen, während sich die Laufzeit eines Kernels einer ungeraden Zeile j ebenfalls über dessen Durchsatz $D_{j\text{te ungerade Zeile}}$ an Hexagonen abschätzen lässt:

$$\begin{aligned} T_{i\text{te gerade Zeile}} &= \frac{N}{D_{i\text{te gerade Zeile}}}, \\ T_{i\text{te ungerade Zeile}} &= \frac{N}{D_{j\text{te ungerade Zeile}}}. \end{aligned} \tag{10.15}$$

Dabei ist N die Anzahl der geraden beziehungsweise der ungeraden Hexagone in jeder Zeile, die sich aus der Systemgröße n und der beschränkten Zugriffsdistanz d wie folgt errechnet:

$$N = \frac{n}{w_{\text{gerade Zeilen}} + w_{\text{ungerade Zeilen}} - (h/2 - 1) \cdot d}. \tag{10.16}$$

Diese theoretischen Überlegungen können wir nun für unseren Ansatz zur Bestimmung der besten Parameter für das hexagonale Tiling-Schema verwenden. Dieser Ansatz gliedert sich in zwei ineinander verschachtelte Suchen:

- **Äußere Suche: Bestimmung der optimalen Höhe der Hexagone und der Verschiebung entlang der Abhängigkeitskette für diese Höhe.** Diese äußere Suche realisierten wir als eine vollständige Suche über die Menge an vielversprechenden Höhen h für die Hexagone und die möglichen Verschiebungen s entlang der Abhängigkeitskette für die jeweilige Höhe. Für jedes Paar von Höhe und Verschiebung muss diese äußere Suche sowohl die beste Breite der geraden Zeilen aus Hexagonen, die beste Breite der ungeraden Zeilen aus Hexagonen und die beste Kernelkonfiguration jeder Zeile bestimmen. Analog wie beim trapezoidalen Tiling-Schema können wir bei dem hexagonalen Tiling-Schema die innere Suche über zwei Ansätze implementieren:

- **1. Ansatz der inneren Suche: Zweidimensionale Suche über die beiden Breiten.** Hierbei nutzen wir wieder aus, dass jeweils die Breite der geraden Zeilen aus Hexagonen und die Breite der ungeraden Zeilen aus Hexagonen einen zweidimensionalen und relativ glatten Konfigurationsraum aufspannen, wodurch wir in diesem Konfigurationsraum mit einem entsprechenden Suchalgorithmus eine performante Konfiguration finden können. Dabei muss diese Suche zusätzlich für jede Konfiguration in diesem zweidimensionalen Konfigurationsraum, die der verwendete Suchalgorithmus abtasten möchte, die optimale Kernelkonfigurationen von den Kernels der geraden Zeilen aus Hexagonen und von den Kernels der ungeraden Zeilen aus Hexagonen bestimmen. Hier verwenden wir wieder eine vollständige Suche über eine Menge von vielversprechenden Kernelkonfigurationen, wobei diese vollständige Suche die Kernelkonfigurationen für jedes Kernel auf Grund der Gleichungen 10.13 sowie 10.14 getrennt voneinander optimieren kann. Somit muss diese zweidimensionale Suche über die beiden Breiten jeweils für jede abzutastende Konfiguration im zweidimensionalen Konfigurationsraum, jede gerade und jede ungerade Zeile aus Hexagonen sowie jede Kernelkonfiguration jeweils ein Kernel für den Zielrechner kompilieren und auf dem Zielrechner ausführen.
- **2. Ansatz der inneren Suche: Benchmarking einer Menge von vielversprechenden Breiten und Kombination der Benchmark-Ergebnisse.** Hier nutzen wir wieder aus, dass der Durchsatz eines Prozessors für sämtliche geraden Zeilen von Hexagonen nur von deren Breite und deren Kernelkonfigurationen, nicht aber von der Breite der ungeraden Zeilen von Hexagonen abhängig ist, und vice versa. Somit können wir für die geraden Zeilen aus Hexagonen für eine Menge von gegebenen Breiten und einer Menge von vielversprechenden Kernelkonfigurationen über eine vollständige Suche für jede Breite jeweils die optimalen Kernelkonfigurationen jeder Zeile und damit den höchsten Durchsatz für diese Breite (siehe Gleichungen 10.11 und 10.16) bestimmen. Analog können wir für die ungeraden Zeilen aus Hexagonen vorgehen. Somit müssen wir für die Bestimmung dieser Durchsätze sowohl für jede Breite, für jede ungerade Zeile aus Hexagonen, für jede gerade Zeile aus Hexagonen und für jede Kernelkonfiguration jeweils ein Kernel kompilieren und ausführen. Anschließend können wir ohne weiteres Benchmarking nur über die Kombination der gemessenen Durchsätze über die Gleichungen 10.13, 10.14 und 10.15 diejenigen Breiten für die geraden und ungeraden Zeilen aus Hexagonen bestimmen, für welche die Laufzeit des hexagonalen Tilings für die aktuelle Kombination von Höhe und Verschiebung der äußeren Suche minimal wird.

Es gilt beim hexagonalen Tiling-Schema aus analogen Gründen wie bei dem trapezoidalen Tiling-Schema wieder, dass sich der 1. Ansatz für die innere Suche besser für CPUs eignet, während sich der 2. Ansatz für die innere Suche besser für GPUs eignet.

Analog, wie bei dem Autotuning-Ansatz für das trapezoidale Tiling-Schema, könnten wir den Autotuning-Ansatz für das hexagonale Tiling-Schema noch weiter durch ein Ausästen des Suchraums über Performance-Modelle verfeinern, so dass er weniger Kernel kompilieren und ausführen muss.

10.2.12 Abbildung der Tiles auf die Threads und die Kerne eines Prozessors

Um die Tiles eines Kernels auf die Threads beziehungsweise auf die Kerne eines Prozessors abzubilden, gibt es folgende Möglichkeiten:

- **Single-Core-Tiling:** Beim *Single-Core-Tiling* wird ein Tile jeweils von einem Kern eines Prozessors berechnet. Das Single-Core-Tiling lässt sich, falls ein Kern per Simultaneous-Multithreading mehrere Threads gleichzeitig ausführen kann, noch einmal unterteilen in:
  - **Single-Thread-Tiling:** Beim *Single-Thread-Tiling* wird ein Tile nur von einem einzigen Thread abgearbeitet.
  - **Multithreaded-Single-Core-Tiling:** Beim *Multithreaded-Single-Core-Tiling* arbeiten die Threads eines Kerns zusammen, um ein Tile zu berechnen.
- **Beim Multi-Core-Tiling:** *Beim Multi-Core-Tiling* wird ein Tile von mehreren unterschiedlichen Kernen abgearbeitet.

Im Folgenden werden wir das Single-Thread-Tiling, das Multithreaded-Single-Core-Tiling und das Multi-Core-Tiling näher vorstellen.

10.2.12.1 Single-Thread-Tiling

Beim *Single-Thread-Tiling* wird jedes Tile jeweils nur von einem einzigen Thread des Prozessors abgearbeitet. Deshalb kann beim Single-Thread-Tiling ein Tile neben dem unter allen Threads geteilten On-Chip-Speicher nur den privaten On-Chip-Speicher seines Threads belegen. Ebenso kann ein Tile nur die Rechenressourcen seines Threads ausnutzen, wodurch ein Mehrkernprozessor für seine Auslastung viele Tiles simultan bearbeiten muss und jedem Tile nur ein kleiner Anteil des unter allen Threads geteilten On-Chip-Speichers zur Verfügung steht. Deshalb übersteigt beim Single-Thread-Tiling der Workingset aller simultan berechneten Tiles bereits bei mittleren Tile-Größen den On-Chip-Speicher des Prozessors, wodurch der Prozessor die Zwischenergebnisse dieser simultanen Tiles in den DRAM auslagern oder deren Eingabevektoren mehrmals aus dem DRAM laden muss. Jedoch müssen bei größeren Zugriffsdistanzen auch die Tiles größer gewählt werden, damit das Tiling überhaupt möglich wird oder das Tiling das Datenvolumen deutlich reduzieren kann. Deshalb ist Single-Thread-Tiling bereits für relativ kleine Zugriffsdistanzen ineffizient. Zusätzlich führt beim Single-Thread-Tiling das Ausnutzen der Simultaneous-Multithreading-Fähigkeit eines Kerns dazu, dass bei n nebenläufigen Threads pro Kern jedem Tile nur $1/n$tel des On-Chip-Speichers zur Verfügung steht.

10.2.12.2 Multithreaded-Single-Core-Tiling

Beim *Multithreaded-Single-Core-Tiling* kooperieren die nebenläufiges Threads eines Kerns an der Berechnung eines Tiles. Somit ist das Multithreaded-Single-Core-Tiling nur auf Architekturen möglich, bei welchen ein Kern per Multithreading mehrere Threads gleichzeitig ausführen kann. Die Vorteile vom Multithreaded-Single-Core-Tiling gegenüber dem Single-Thread-Tiling unterscheiden sich je nachdem, ob der Prozessor eine Multithreading-Implementierung besitzt, bei welcher sich die Threads eines Kerns den On-Chip-Speicher und die Rechenwerke teilen, oder bei welcher die Threads eines Kerns jeweils ihre eigenen replizierten On-Chip-Speicher und Rechenwerke besitzen:

- **Multithreading ohne replizierten On-Chip-Speicher oder mit nur wenig replizierten On-Chip-Speicher und ohne replizierte Rechenwerke:** Auf solchen Architekturen kann Single-Thread-Tiling mit einem Tile beziehungsweise Thread pro Kern bereits einen Großteil des privaten On-Chip-Speichers eines Kerns belegen und potentiell die gesamte Rechenleistung des Kerns ausnutzen. Führt Single-Thread-Tiling mehr als ein Tile pro Kern aus, kann der Kern so nur potentiell die Latenzen des Tiles besser überbrücken, wodurch er die Auslastung seiner Rechenwerke potentiell erhöhen kann. Allerdings reduziert sich dadurch wiederum der pro Tile zur Verfügung stehende On-Chip-Speicher. Im Gegensatz dazu kann Multithreaded-Single-Core-Tiling das Multithreading eines Kerns zur besseren Latenzüberbrückung ausnutzen, ohne mehr als ein Tile pro Kern gleichzeitig berechnen zu müssen, wodurch sich durch die Ausnutzung des Multithreadings der für jedes Tile zur Verfügung stehende On-Chip-Speicher nicht reduziert.
- **Multithreading mit viel replizierten On-Chip-Speicher und replizierten Rechenwerken:** Auf solchen Architekturen muss das Single-Thread-Tiling mehr als ein Tile pro Kern gleichzeitig ausführen, um den gesamten privaten On-Chip-Speicher und die gesamte Rechenleistung eines Kerns auszunutzen. Dadurch erzielt das Multithreaded-Single-Core-Tiling gegenüber dem Single-Thread-Tiling dieselben Vorteile wie das Multi-Core-Tiling: Der Prozessor muss für seine Saturierung beim Multithreaded-Single-Core-Tiling gegenüber dem Single-Core-Tiling weniger Tiles gleichzeitig berechnen, wodurch jedes Tile wiederum einen größeren Anteil des On-Chip-Speichers belegen kann.

Gegenüber dem Multi-Core-Tiling kann das Multithreaded-Single-Core-Tiling punkten, dass sich die Threads eines Kerns mit geringerem Overhead synchronisieren und über den privaten On-Chip-Speicher des Kerns effizienter Zwischenergebnisse austauschen können. Jedoch benötigt das Multithreaded-Single-Core-Tiling für die Saturierung des Prozessors wiederum auch mehr nebenläufige Tiles als das Multi-Core-Tiling.

10.2.12.3 Multi-Core-Tiling

Beim *Multi-Core-Tiling* kooperieren mehrere Kerne des Prozessors an der Berechnung eines Tiles. Dadurch kann ein Tile auch den privaten On-Chip-Speicher von mehreren Kernen belegen und die Rechenleistung von mehreren Kernen ausnutzen. Ebenso muss der Prozessor weniger Tiles gleichzeitig berechnen, um ausgelastet zu sein, wodurch jedes Tile wiederum einen größeren Anteil des geteilten On-Chip-Speichers belegen kann. Mehr On-Chip-Speicher für ein Tile bedeutet auch, dass beim Multi-Core-Tiling größere Tile-Größen wiederum effizienter werden, wodurch das Multi-Core-Tiling wiederum für größere Zugriffsdistanzen effizienter wird. Offensichtlich können wir die Anzahl an Kernen pro Tile so wählen, dass alle Kerne des Prozessors an einem einzigen Tile zusammenarbeiten. Jedoch können wir auch eine kleinere Anzahl an Kernen pro Tile wählen, so dass der Prozessor mehrere Tiles gleichzeitig bearbeiten kann. Wählen wir weniger Kerne pro Tile, so reduzieren wir damit den Overhead der Synchronisation und erzeugen eine weniger stark synchronisierte Instruktionsmischung während der Ausführung, wodurch sich wiederum die Ausnutzung der Ressourcen des Prozessors erhöhen kann. Jedoch benötigt Multi-Core-Tiling bei weniger Kernen pro Tile auch mehr parallele Tiles, um den Prozessor zu saturieren, wodurch sich wiederum auch die effizienten Tile-Größen reduzieren und das Datenvolumen sowie das DRAM-Volumen erhöhen. Deshalb erwarten wir, dass viele kleine nebenläufige Tiles mit wenigen Kernen pro Tile bei kleinen Zugriffsdistanzen

die beste Laufzeit erzielen, während wenige große Tiles mit vielen Kernen pro Tile bei großen Zugriffsdistanzen die beste Laufzeit erzielen.

10.2.12.4 Traditionelle Ansätze für das Tiling auf unterschiedlichen Architekturen

Je nach Architektur ist das Single-Thread-Tiling oder das Multithreaded-Single-Core-Tiling der traditionelle Ansatz, um ein Tiling zu implementieren:

- **CPU-Architekturen (mit Multithreading):** Auf solchen CPU-Architekturen täuscht das Betriebssystem für jeden physikalischen Kern mehrere logische Kerne vor, wobei jeder dieser logischen Kerne nur einen einzigen Hardware-Thread gleichzeitig ausführen kann. Dadurch können Anwendungen den logischen Kernen Aufgaben beziehungsweise Threads zuweisen, so als ob diese vollwertige Kerne wären. Darüber können Anwendungen wiederum die Multithreading-Fähigkeit der CPU ausnutzen, ohne auf diese explizit eingehen zu müssen. Aus diesem Grund weist man im traditionellen Ansatz für Tiling auf CPUs, wie er zum Beispiel von der Arbeit [Z24] und [K24] verwendet wird, jedem virtuellen Kern seine eigenen Tiles zu, womit dieser Ansatz dem Single-Thread-Tiling entspricht.
- **GPU-Architekturen:** Auf GPUs startet man beim traditionellen Ansatz für das Tiling, wie er zum Beispiel von den Arbeiten [K20] und [K29] verwendet wird, eine Workgroup pro Tile. Da eine Workgroup jedoch von den nebenläufigen Threads eines Kerns abgearbeitet wird, entspricht dieser Ansatz bereits dem Multithreaded-Single-Core-Tiling.

10.3 Tiling auf GPUs

10.3.1 Implementierung

10.3.1.1 Allgemeines

Um das Tiling auf GPUs zu implementieren, übersetzen wir wieder jedes abstrakte Kernel aus unseren theoretischen Überlegungen in ein 1D-GPU-Kernel. Wir implementierten sowohl das Multithreaded-Single-Core-Tiling als auch das Multi-Core-Tiling. Bei dem Ersteren entspricht ein Tile jeweils einer Workgroup in einem GPU-Kernel, weshalb wir es im Folgenden als *Single-Workgroup-Tiling* bezeichnen (siehe Abbildung 10.8). Beim Multi-Core-Tiling arbeiten in einem GPU-Kernel jeweils mehrere aufeinanderfolgende simultan ausgeführte Workgroups auf unterschiedlichen GPU-Kernen an einem Tile zusammen, weshalb wir es im Folgenden als *Multi-Workgroup-Tiling* (siehe Abbildung 10.9) bezeichnen. Sowohl beim Single-Workgroup-Tiling als auch beim Multi-Workgroup-Tiling berechnen die Workitems beziehungsweise die Workgroups eines Tiles die Komponenten einer Vektorgrundoperation des Tiles parallel zueinander, während sie die Vektorgrundoperationen in der Abhängigkeitskette des Tiles nacheinander berechnen.

Wir wählen für die Varianten mit Tiling wieder, wie bei den Varianten mit Kernelfusion, für alle Vektorgrundoperationen das gleiche $n : 1$ Mapping zwischen Vektorkomponenten und Workitems, um eine Datenwiederverwendung über den Registersatz der GPU zu ermöglichen. Da eine GPU ihren Registersatz nicht dynamisch indexieren kann, müssen wir dieses $n : 1$ Mapping wieder ohne Schleifen realisieren. Durch dieses Mapping kann auf der GPU ein Tile, analog zu einem Block in einer Variante mit Kernelfusion, Vektorkomponenten einmal aus dem DRAM in die Register der GPU laden, und dann diese Register für mehrere fusionierte Vektorgrundoperationen wiederverwenden. Auch treten analog wie bei den Varianten mit Kernelfusion bei fusionierten (RHS | LC) $\rightarrow$ (LC | MAP | RED)-Abhängigkeiten in einem Tile keine Abhängigkeiten zwischen den Workitems des Tiles auf, wodurch das Tile diese Abhängigkeiten nur über den Registersatz verschmelzen kann. Dahingegen treten bei einer fusionierten LC $\rightarrow$ RHS-Abhängigkeit in einem Tile eben solche Abhängigkeiten zwischen den unterschiedlichen Workitems beziehungsweise Workgroups des Tiles auf, wodurch bei einer fusionierten LC $\rightarrow$ RHS-Abhängigkeit eine Synchronisation und ein Datentransfer zwischen den Workitems beziehungsweise Workgroups des Tiles nötig ist. Dieser Datentransfer für fusionierte LC $\rightarrow$ RHS-Abhängigkeiten findet beim Single-Workgroup-Tiling über einen Buffer im Scratchpadspeicher des Kerns der jeweiligen Workgroup und beim Multi-Workgroup-Tiling über einen Buffer im Heap-Speicher der GPU statt.

Da das Mapping zwischen den Workitems und den Vektorkomponenten für alle Vektorgrundoperationen in einem Tile identisch ist, aber ein Tile für unterschiedliche Vektorgrundoperationen unterschiedliche Argumentintervalle besitzt, müssen wir vor jeder Vektorgrundoperation per Predication oder Sprungbefehl sicherstellen, dass die Workitems nur diejenigen Komponenten berechnen, die sich innerhalb des Argumentintervalls der Vektorgrundoperation befinden. Analog müssen wir über Predication oder Sprungbefehle sicherstellen, dass die Workitems des Tiles nur diejenigen Komponenten von einem permanenten Vektor aus dem DRAM lesen oder in den DRAM schreiben, die das Tile auch tatsächlich lesen beziehungsweise schreiben soll.

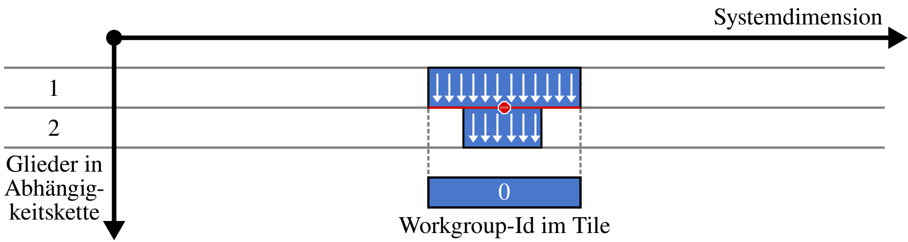


Abbildung 10.8: Single-Workgroup-Tiling

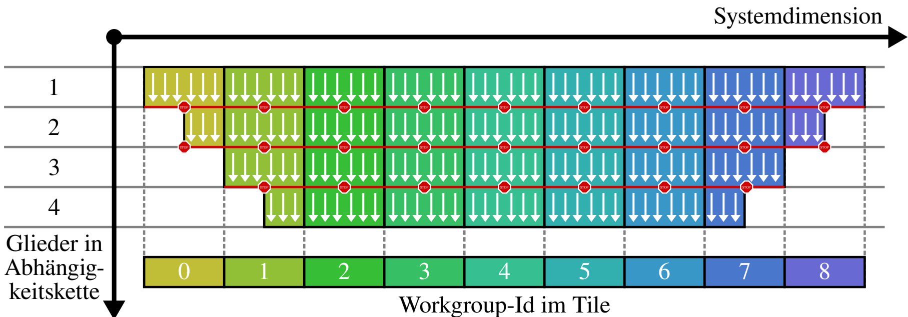


Abbildung 10.9: Multi-Workgroup-Tiling

10.3.1.2 Fusion von LC → RHS-Abhängigkeiten

Als Nächstes werden wir vorstellen, wie ein GPU-Kernel mit Tiling eine fusionierte LC → RHS-Abhängigkeit umsetzen muss. Denn bei einer fusionierten LC → RHS-Abhängigkeit treten in einem Tile Abhängigkeiten zwischen den unterschiedlichen Workitems des Tiles auf, wodurch bei einer solchen fusionierten LC → RHS-Abhängigkeit eine Synchronisation und ein Datentransfer zwischen den Workitems des Tiles nötig ist. Für eine fusionierte LC → RHS-Abhängigkeit unterscheiden sich sowohl die Synchronisation als auch der Datentransfer innerhalb eines Tiles je nach Tiling-Variante:

- **Synchronisation und Datentransfer für LC → RHS-Abhängigkeiten beim Single-Workgroup-Tiling:** Beim Single-Workgroup-Tiling existieren in einem Tile bei fusionierten LC → RHS-Abhängigkeiten nur Abhängigkeiten zwischen den Workitems innerhalb der Workgroup des Tiles. Dadurch kann das Tile für solche fusionierten LC → RHS-Abhängigkeiten den Scratchpad-Speicher des Kerns für einen schnellen Datentrans-

fer zwischen seinen Workitems und eine Workgroup-weite Hardware-Barriere für die schnelle Synchronisation seiner Workitems verwenden. Diese Barriere bezeichnen wir im folgenden als *Single-Workgroup-Barriere*.

- **Synchronisation und Datentransfer für LC $\rightarrow$ RHS-Abhängigkeiten beim Multi-Workgroup-Tiling:** Beim Multi-Workgroup-Tiling existieren in einem Tile bei fusionierten LC $\rightarrow$ RHS-Abhängigkeiten auch Abhängigkeiten zwischen den Workitems aus den unterschiedlichen Workgroups des Tiles. Somit muss bei einem Multi-Workgroup-Tile der Datentransfer zwischen den Workitems beziehungsweise den Workgroup des Tiles für fusionierte LC $\rightarrow$ RHS-Abhängigkeiten über den langsameren Heap-Speicher, das heißt effektiv über den L2-Cache oder den DRAM, erfolgen. Ebenso existiert auf GPUs keine schnelle Hardware-Barriere, um mehrere Workgroups in einem Kernel miteinander zu synchronisieren, wodurch das Multi-Workgroup-Tiling bei einer fusionierten LC $\rightarrow$ RHS-Abhängigkeit eine langsame, mehrere Workgroups umspannende Software-Barriere benötigt. Diese Barriere bezeichnen wir im folgenden als *Multi-Workgroup-Barriere*.

Deshalb muss sowohl ein Single-Workgroup-Tile als auch ein Multi-Workgroup-Tile bei einer verschmolzenen LC $\rightarrow$ RHS-Abhängigkeit folgende Schritte durchlaufen:

1. **Schreiben des Ergebnisses der LC-Operation in den Buffer:** Die Workitems des Tiles schreiben diejenigen Komponenten von der LC-Operation, die das Tile selbst produziert und somit in den Registern abgespeichert hat, und die das Tile benötigt, um die RHS-Operation zu berechnen, in einen Buffer, der beim Single-Workgroup-Tiling innerhalb des schnelleren Scratchpad-Speichers und beim Multi-Workgroup-Tiling innerhalb des langsameren Heap-Speichers liegt.
2. **Synchronisation über eine Barriere:** Die Workitems des Tiles führen eine Barriere aus, um sich untereinander zu synchronisieren und um sicherzustellen, dass alle Schreibvorgänge in den Buffer abgeschlossen und sichtbar sind. Bei dieser Barriere handelt es sich um eine schnellere Single-Workgroup-Barriere beim Single-Workgroup-Tiling und eine langsamere Multi-Workgroup-Barriere beim Multi-Workgroup-Tiling.
3. **Lesen aus dem Buffer beim Berechnen der RHS-Operation:** Die Workitems des Tiles berechnen die RHS-Operation und lesen für diesen Zweck aus dem Buffer, das heißt aus dem Scratchpadspeicher beim Single-Workgroup-Tiling und aus dem Heap-Speicher beim Multi-Workgroup-Tiling. Produziert ein Tile mindestens all diejenigen Komponenten der LC-Operation, die es für die Berechnung der RHS-Operation benötigt, so befinden sich auch all diese Komponenten in dem Buffer. In demjenigen Fall, dass ein Tile bei der LC-Operation nicht alle Komponenten produziert, die es für die Auswertung der RHS-Operation benötigt, befinden sich nur ein Teil der benötigten LC-Komponenten in dem Buffer, während sich die verbleibenden Komponenten in einem permanenten Vektor im Heap-Speicher befinden.

Aus diesem Grund benötigen wir für das Tiling zwei unterschiedliche Implementierungen der RHS-Operation, die wir je nach der Position der RHS-Operation innerhalb eines Tiles aufrufen müssen:

- **RHS-Consecutive:** In der Implementierung namens *RHS-Consecutive* liegen alle Vektorkomponenten aufeinanderfolgend in einem Speicherbereich, welcher ein permanenter

Vektor im Heap-Speicher oder ein Buffer für die Verschmelzung einer LC $\rightarrow$ RHS-Abhängigkeit sein kann.
- **RHS-Split:** In der Implementierung namens *RHS-Split* gibt es zwei unterschiedliche Speicherbereiche. Ein Speicherbereich ist der rechte und linke Rand des Tiles, welcher ein permanenter Vektor im Heap-Speicher beziehungsweise im DRAM ist, während der andere Speicherbereich die Mitte des Tiles ist, wodurch dieser Speicherbereich einem Scratchpad-Buffer oder Heap-Buffer für die Verschmelzung einer LC $\rightarrow$ RHS-Abhängigkeit entspricht.

Als Alternative hierzu ist es auch möglich, bei einer fusionierten LC $\rightarrow$ RHS-Abhängigkeit diejenigen Komponenten der LC-Operation, welche von der RHS-Operation benötigt werden, in einen Buffer zu kopieren. Dadurch wird die RHS-Split-Implementierung der RHS-Operation nicht mehr benötigt. Jedoch vergrößert sich dafür gerade beim Multi-Workgroup-Tiling auch der Workingset und der Overhead.

In einem Tile, bei welchem mehr als zwei Glieder der Abhängigkeitskette miteinander verschmolzen sind, gibt es zwei unterschiedliche Ansätze für das Buffering beim Datentransfer entlang einer LC $\rightarrow$ RHS-Abhängigkeit:

- **Single-Buffering:** Beim *Single-Buffering* besitzt das Tile nur einen einzigen Buffer für den Datentransfer bei LC $\rightarrow$ RHS-Abhängigkeiten. Dadurch benötigt ein Tile für jedes verschmolzenes Glied der Abhängigkeitskette zwei Barrieren. Dabei stellt das Tile durch die erste Barriere sicher, dass all seine Workitems mit dem Lesen aus dem Buffer abgeschlossen haben, bevor seine Workitems in den Buffer hineinschreiben. Analog stellt das Tile durch die zweite Barriere sicher, dass all seine Workitems mit dem Schreiben in den Buffer abgeschlossen haben, bevor seine Workitems aus dem Buffer lesen.
- **Double-Buffering:** Beim *Double-Buffering* besitzt ein Tile zwei alternierende Buffer für den Datentransfer bei LC $\rightarrow$ RHS-Abhängigkeiten. Während für ein RHS $\rightarrow$ LC-Glied der eine Buffer dem Tile als Lesebuffer und der andere als Schreibbuffer dient, vertauscht das Tile die beiden Buffer nach der Barriere, so dass für das nächste RHS $\rightarrow$ LC-Glied der frühere Schreibbuffer nun der Lesebuffer und der frühere Lesebuffer nun der Schreibbuffer des Tiles ist.

Somit benötigt ein Tile beim Single-Buffering zwei Barrieren pro RHS $\rightarrow$ LC-Glied und beim Double-Buffering nur eine einzige Barriere pro RHS $\rightarrow$ LC-Glied, wodurch Single-Buffering einen größeren Synchronisationsoverhead als Double-Buffering verursacht. Da aber ein Tile beim Single-Buffering nur einen einzigen Buffer und beim Double-Buffering zwei alternierende Buffer benötigt, besitzt Single-Buffering einen geringeren Workingset als Double-Buffering. Da beim Single-Workgroup-Tiling die Single-Workgroup-Barriere sehr günstig ist, während beim Multi-Workgroup-Tiling die Multi-Workgroup-Barriere einen großen Overhead besitzt, ist anzunehmen, dass für Single-Workgroup-Tiling Single-Buffering und für Multi-Workgroup-Tiling Double-Buffering eine bessere Performance liefert. Durch eine experimentelle Überprüfung konnten wir diese Annahme bestätigen.

Um beim Multi-Workgroup-Tiling den Workingset für das Buffering zu reduzieren, implementierten wir zwei weitere Optimierungen, durch welche ein Tile den Overhead und den Workingset bei einer fusionierten LC $\rightarrow$ RHS-Abhängigkeit reduzieren kann:

- Falls ein Tile bei einer fusionierten LC → RHS-Abhängigkeit bereits alle Komponenten der LC-Operation für ein späteres Kernel zum Lesen in den permanenten Vektor beziehungsweise in den DRAM zurückschreibt, kann das Tile in seinem Inneren die Daten entlang dieser fusionierten LC → RHS-Abhängigkeit auch über diesen permanenten Vektor transferieren. Dadurch benötigt das Tile keinen designierten Buffer für den Datentransfer entlang dieser LC → RHS-Abhängigkeit.
- Falls ein Tile nur die Komponenten der LC-Operation am rechten und linken Rand des Tiles für ein späteres Kernel zum Lesen in den permanenten Vektor zurückschreibt, kann das Tile in seinem Inneren zumindest diese Komponenten am Rand des Tiles über den permanenten Vektor transferieren, während es nur noch die Komponenten in der Mitte des Tiles über einen Buffer transferieren muss.

10.3.1.3 Datenwiederverwendung beim Tiling auf GPUs

Ein Tile kann auf einer GPU deren On-Chip-Speicher, abgesehen von der Datenwiederverwendung bei einer fusionierten LC → RHS-Abhängigkeit, analog wie ein Block bei einer Variante mit Fusion ausnutzen:

- **Datentransfer für (RHS | LC | MAP) → (LC | MAP | RED)-Abhängigkeiten über den Registersatz:** Das Tile verwendet den Registersatz der GPU, um Daten entlang fusionierten (RHS | LC | MAP) → (LC | MAP | RED)-Abhängigkeiten zu transferieren. Hier kann aber innerhalb eines Tiles derjenige Sonderfall auftreten, dass nur die Workitems in der Mitte des Tiles die Komponenten der (RHS | LC | MAP)-Operation am Anfang der Abhängigkeit selbst berechnen und in den Registern abspeichern, während die Workitems am Rand des Tiles die Komponenten dieser der (RHS | LC | MAP)-Operation aus dem DRAM in die Register laden.
- **Datentransfer für LC → RHS-Abhängigkeit über den Scratchpad-Speicher beim Single-Workgroup-Tiling beziehungsweise über den L2-Cache beim Multi-Workgroup-Tiling:** Das Tile verwendet beim Single-Workgroup-Tiling den Scratchpadspeicher und beim Multi-Workgroup-Tiling den L2-Cache der GPU, um Daten entlang einer fusionierten LC → RHS-Abhängigkeit zu übertragen. Dabei kann jedoch wieder derjenige Sonderfall auftreten, dass die Workitems am Rand des Tiles für die Berechnung der RHS-Operation auch diejenigen Komponenten der LC-Operation benötigen, die nicht von dem Tile selbst, sondern von einem vorherigen Kernel produziert wurden und deshalb von dem Tile aus dem DRAM geladen werden müssen.
- **Wiederverwendung von Registern, um ein redundantes Laden aus dem DRAM zu vermeiden:** Das Tile verwendet die Register der GPU, um redundante Ladeoperationen zu vermeiden, wenn es mehrere verschmolzene (RHS | MAP | LC | RED)-Operationen berechnen muss, die alle auf denselben permanenten Argumentvektor im DRAM zugreifen. Hier kann aber innerhalb eines Tiles wieder derjenige Sonderfall auftreten, dass nur die Workitems am Rand des Tiles die Komponenten aus dem DRAM in die Register laden, während die Workitems in der Mitte des Tiles die Komponenten selbst berechnen.
- **Datenwiederverwendung über die Caches, um ein redundantes Laden eines permanenten Vektors aus dem DRAM zu vermeiden:** Das Tile verwendet die Caches

der GPU, um die Komponenten eines permanenten Vektors aus dem DRAM wiederzuverwenden, welchen das Tile für eine RHS-Operation und für eine oder mehrere LC-Operationen als Argument benötigt. Denn während in diesem Fall die beschränkte Zugriffsdistanz der RHS-Operation keine Datenwiederverwendung ausschließlich über den Registersatz ermöglicht, so kann das Tile dennoch ausnutzen, dass wegen der beschränkten Zugriffsdistanz all seine Workitems für die Auswertung der RHS-Operation lokal auf den permanenten Vektor zugreifen. Dadurch lädt das Tile bei der Auswertung der RHS-Operation diejenigen Komponenten des permanenten Vektors in die Caches der GPU, die es auch für die Auswertung der LC-Operationen benötigt. Bei der Berechnung der LC-Operation durch das Tile werden dann die Zugriffe auf den permanenten Vektor durch die Caches abgefangen, wodurch das Tile den permanenten Vektor insgesamt nur ein einziges Mal aus dem DRAM laden muss.

- **Kein unnötiges Zurückschreiben von Registern in den DRAM:** Das Tile schreibt nur diejenigen produzierten Komponenten einer (RHS | LC | MAP)-Operation von den Registern in den dazugehörigen permanenten Vektor und damit in den DRAM zurück, die auch von einem späteren Kernel gelesen werden, das heißt entweder alle der im Tile produzierten Komponenten, nur die produzierten Komponenten an dem rechten oder linken Rand des Tiles oder keine der im Tile produzierten Komponenten.

Dabei gilt wieder wie bereits bei der Kernelfusion, dass die Datenwiederverwendung über den Registersatz auf GPUs sehr vielversprechend ist, da er mit Abstand der größte und schnellste On-Chip-Speicher ist. Jedoch kann in Abhängigkeit von der Struktur des ODE-Verfahrens entweder die Größe des Registersatzes oder die Größe des Scratchpad-Speichers beim Single-Workgroup-Tiling beziehungsweise die Größe des L2-Caches beim Multi-Workgroup-Tiling die Datenwiederverwendung limitieren:

- Bei ODE-Verfahren, deren LC-Operationen viele Argumentvektoren besitzen, zum Beispiel RK-Verfahren mit hoher Ordnung, findet die Datenwiederverwendung primär über den Registersatz statt, wodurch die effektive Tile-Größe und damit die Datenwiederverwendung primär durch die Größe des Registersatzes limitiert ist.
- Bei ODE-Verfahren, deren LC-Operationen nur wenige Argumentvektoren besitzen, zum Beispiel RK-Verfahren mit niedriger Ordnung, wird die effektive Tile-Größe und damit die Datenwiederverwendung beim Single-Workgroup-Tiling durch die Größe des Scratchpad-Speichers und beim Multi-Workgroup-Tiling durch die Größe des L2-Caches limitiert.

10.3.1.4 Multi-Workgroup-Barrieren für das Multi-Workgroup-Tiling

10.3.1.4.1 Implementierungsvarianten für eine Multi-Workgroup-Barriere

Um die Synchronisation für eine fusionierte LC $\rightarrow$ RHS-Abhängigkeit in einem Multi-Workgroup-Tile zu realisieren, benötigen wir eine Multi-Workgroup-Barriere. Unglücklicherweise existiert auf modernen GPUs nur eine Hardware-Barriere zwischen den Threads beziehungsweise zwischen den Workitems einer einzigen Workgroup (Single-Workgroup-Barriere).

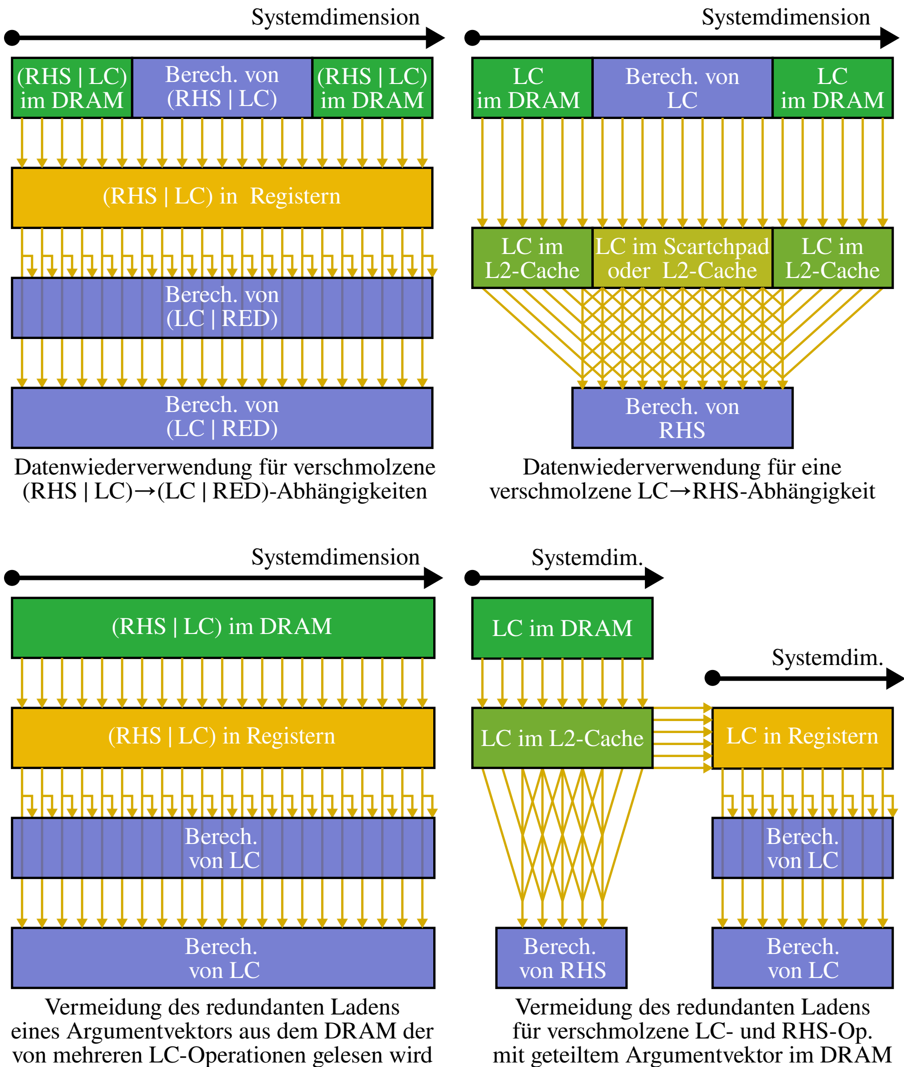


Abbildung 10.10: Datenwiederverwendung in einem Tile auf der GPU. Dabei gilt, dass das Single-Workgroup-Tiling für den Datentransfer entlang einer LC → RHS-Abhängigkeit den Scratchpad-Speicher verwendet, während das Multi-Workgroup-Tiling für den Datentransfer entlang einer LC → RHS-Abhängigkeit den L2-Cache verwendet.

Zudem ist es auf GPUs nicht möglich, zwei Workgroups über Signale miteinander zu synchronisieren. Folglich müssen wir die Multi-Workgroup-Barriere als eine Software-Barriere mit Busy-Waiting implementieren. Wir implementierten zwei Varianten für diese Multi-Workgroup-Barriere aus der Arbeit [K18], und betrachten zudem die Barriere aus der Cooperative-Groups-Bibliothek (siehe [T9]) von NVIDIA:

- **Barriere vom Typ-ATOMIC (Barriere mit einem geteilten atomaren Zähler):** Diese *Barriere vom Typ-ATOMIC* basiert auf einem atomaren Zähler, welchen sich alle Workgroups der Barriere untereinander teilen. Kommt eine Workgroup an der Barriere an, so inkrementiert sie zunächst diesen Zähler per atomarer Operation um eins. Anschließend wartet die Workgroup durch ein dauerhaftes Abfragen des Zählers in einer Schleife darauf, dass auch alle anderen Workgroups an der Barriere angekommen sind, bevor sie die Barriere verlässt. Um zu vermeiden, dass der atomare Zähler zum Flaschenhals wird, implementierten wir diese Barriere so, dass der Druck auf den atomaren Zähler möglichst gering ist. Somit führt in unserer Implementierung nur ein designiertes Master-Workitem einer Workgroup die atomare Operation auf diesen atomaren Zähler und das Busy-Waiting auf ebendiesen aus, während die übrigen Workitems der Workgroup sich mit dem Master-Workitem über Single-Workgroup-Barrieren beim Betreten und beim Verlassen der Multi-Workgroup-Barriere synchronisieren.
- **Barriere vom Typ-MASTER (Barriere mit einer Master Workgroup):** In dieser *Barriere vom Typ-MASTER* existiert eine designierte *Master-Workgroup*, welche überprüft, ob bereits alle anderen Workgroups, im Folgenden *Slave-Workgroups* genannt, an der Barriere angekommen sind, während die Slave-Workgroups nach dem Ankommen an der Barriere auf das Signal der Master-Workgroup warten, dass sie die Barriere verlassen dürfen. Dabei erfolgt die Kommunikation zwischen der Master- und den Slave-Workgroups in dieser Barriere über *Flags*, wobei jede Slave-Workgroup ihre eigene *Betreten-Flag* und *Verlassen-Flag* besitzt: Betritt eine Slave-Workgroup die Barriere, so setzt sie zunächst ihre Betreten-Flag von falsch auf wahr. Dann wartet die Slave-Workgroup per Busy-Waiting in einer Schleife darauf, dass ihre Verlassen-Flag von der Master-Workgroup von falsch auf wahr gesetzt wird. In der Zwischenzeit wartet die Master-Workgroup ebenfalls per Busy-Waiting darauf, dass jede Slave-Workgroup ihre Betreten-Flag von falsch auf wahr gesetzt hat. Nachdem dies eingetreten ist, beginnt die Master-Workgroup damit, die Verlassen-Flags der Slave-Workgroups von falsch auf wahr zu setzen, so dass die Slave-Workgroups die Barriere verlassen können.
- **Barriere vom Typ-NVIDIA (Barriere aus Cooperative-Groups-Bibliothek von NVIDIA):** Ein Kernel kann eine *Barriere vom Typ-NVIDIA* nur dafür verwenden, um alle seine Workgroups untereinander zu synchronisieren. Folglich erlaubt diese Barriere nur, dass die GPU ein einziges Multi-Workgroup-Tile gleichzeitig ausführt. Zusätzlich benötigt sie einen persistenten Threading-Ansatz, bei welchem das Kernel nur so viele Workgroups startet, wie die GPU gleichzeitig ausführen kann. Des Weiteren ist diese Barriere nur für Pascal-, Volta- und Turing-GPUs verfügbar. Während NVIDIA nicht die Implementierungsdetails dieser Barriere dokumentiert, so lässt ihre schlechte Performance vermuten, dass es sich bei ihr um eine nicht optimal implementierte Software-Barriere handelt.

Da der L2-Cache einer GPU aus mehreren parallelen Partitionen besteht, wobei jede Partition eine exklusive Lese-, Schreib- und Atomic-Policy besitzt, sequentialisiert bei einer Barriere

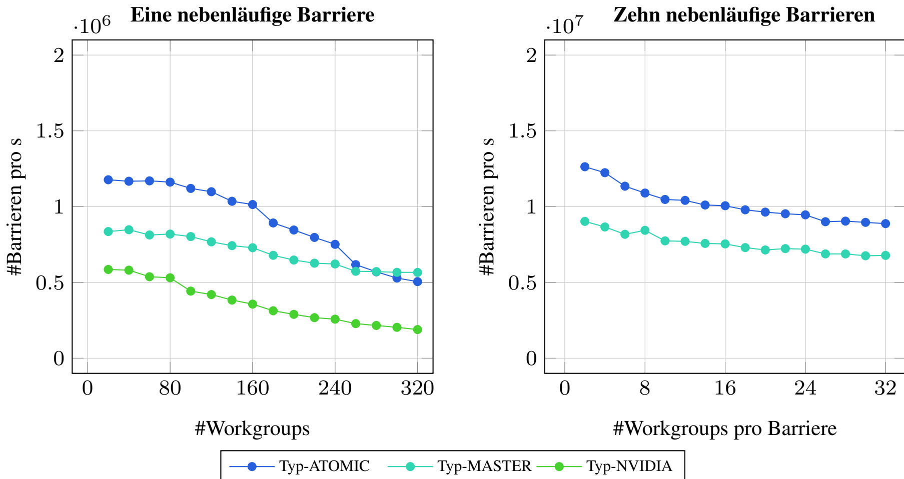


Abbildung 10.11: Mikro-Benchmark der Multi-Workgroup-Barrieren auf einer Volta-GPU. In diesem Mikro-Benchmark führten wir für die unterschiedlichen Multi-Workgroup-Barrieren jeweils mit einer nebenläufigen Barriere (links) und zehn nebenläufigen Barrieren (rechts) eine Messreihe durch. Für die Diagramme wählten wir für beide Achsen jeweils eine unterschiedliche Skalierung.

vom Typ-ATOMIC der L2-Cache sowohl das Inkrementieren als auch das Lesen des atomaren Zählers. Jedoch kann ein Kernel auf die atomaren Zähler von mehreren nebenläufigen Barrieren vom Typ-ATOMIC parallel zugreifen, sofern diese Zähler untereinander einen passenden Stride besitzen, so dass sie sich in unterschiedlichen Partitionen des L2-Caches befinden. Während innerhalb einer einzigen Barriere vom Typ-ATOMIC keine Parallelität existiert, so können die Flags bei einer einzigen Barriere vom Typ-MASTER bereits parallel gesetzt und abgefragt werden. Dies setzt allerdings wieder voraus, dass die Flags sich auf Grund eines passenden Strides in den unterschiedlichen Partitionen des L2-Caches befinden. Des Weiteren besitzt eine Barriere vom Typ-MASTER eine höhere Latenz als eine Barriere von Typ-ATOMIC, da bei einer Barriere vom Typ-ATOMIC die Synchronisation direkt über den L2-Cache erfolgt, während bei einer Barriere vom Typ-MASTER der L2-Cache zuerst von der Master-Workgroup gelesen und dann beschrieben werden muss. Somit erwarten wir, dass eine Barriere vom Typ-ATOMIC für Tiles, die aus wenigen Workgroups bestehen, besser geeignet ist, während eine Barriere vom Typ-MASTER für Tiles, die aus vielen Workgroups bestehen, besser geeignet ist.

10.3.1.4.2 Micro-Benchmark für die Implementierungsvarianten

Um die Performance und die Skalierung der drei Implementierungsvarianten für eine Multi-Workgroup-Barriere miteinander zu vergleichen, entwickelten wir kein kleines Micro-Benchmark. Dieses Benchmark misst den Durchsatz von n simultanen Barrieren mit m Workgroups pro Barriere. Dafür startet das Micro-Benchmark zunächst $n \cdot m$ Workgroups. Anschließend ruft jede Workgroup eintausend-mal die ihr zugewiesene Barriere auf, um an ihr zu warten. Dabei gilt anzumerken, dass eine GPU diese n Barrieren auch nur wirklich simultan aus-

führt, wenn sie mindestens $n \cdot m$ Workgroups gleichzeitig ausführen kann. Die Ergebnisse des Micro-Benchmarks auf einer Geforce Titan V (*Volta*) mit 80 Kernen zeigen wir in Abbildung Abbildung 10.11 ein.

Mit dem Micro-Benchmark können wir folgende Fragen beantworten:

Welche Barriere ist besser? Die Messungen zeigen, wie erwartet, dass eine Barriere vom Typ-ATOMIC einen geringeren Overhead als eine Barriere vom Typ-MASTER besitzt, dafür aber auch deutlich schlechter skaliert. Folglich ist eine Barriere vom Typ-ATOMIC bei weniger als 300 Workgroups pro Barriere schneller, während ab 300 Workgroups pro Barriere eine Barriere vom Typ-MASTER schneller ist. Mehr als 300 Workgroups pro Barriere wären aber beim Multi-Workgroup-Tiling auf einer Volta-GPU nur möglich, wenn sie nur ein einziges Multi-Workgroup-Tile mit mindestens 4 Workgroups per Kern gleichzeitig ausführen würde. Da ein solches Tile mit so vielen Workgroups pro Kern wegen des hohen Synchronisationsoverheads sehr ineffizient ist, ist auf der Volta-GPU die Barriere vom Typ-ATOMIC für das Multi-Workgroup-Tiling immer besser geeignet als die Barriere vom Typ-MASTER. Obwohl man erwarten würde, dass die Barriere vom Typ-NVIDIA eine gute Performance besitzt, ist sie deutlich langsamer als eine Barriere vom Typ-ATOMIC oder Typ-MASTER.

Ist eine Barriere vom Typ-ATOMIC durch die Latenz oder durch den Durchsatz des atomaren Zählers gebunden? Die Performance einer Barriere vom Typ-ATOMIC ist bis ungefähr 160 Workgroups pro Barriere nahezu konstant. Erst bei mehr als 160 Workgroups pro Barriere fällt die Performance der Barriere deutlich ab. Dies deutet darauf hin, dass die Barriere mit bis zu 160 Workgroups durch die Latenz des atomaren Zählers gebunden ist, und erst bei mehr als 160 Workgroups pro Barriere durch den Durchsatz des atomaren Zählers limitiert wird.

Wie skaliert die Performance mit der Anzahl an nebenläufigen Barrieren? Auf der Volta-GPU besitzen zehn nebenläufige Barrieren in etwa einen zehnmal so großen Durchsatz wie nur eine einzige nebenläufige Barriere. Ebenso ist die Performance in dem Benchmark mit 10 nebenläufigen Barrieren nahezu unabhängig von der Anzahl an Workgroups pro Barriere. Aus beiden kann man folgern, dass sogar für viele nebenläufige Barrieren das Speichersubsystem der GPU nicht zum Flaschenhals wird.

Wie schneiden unsere Implementierungen einer Multi-Workgroup-Barriere im Vergleich zu den Implementierungen einer Multi-Workgroup-Barriere in anderen Arbeiten ab? Leider gibt es keine uns bekannten aktuellen Arbeiten, die Multi-Workgroup-Barrieren auf modernen GPUs implementieren und untersuchen. Die Experimente in der Arbeit [K18] aus dem Jahre 2010 zeigen jedoch, dass auf einer Geforce 280 GTX eine Barriere vom Typ-MASTER bei bereits mehr als vier Workgroups pro Barriere schneller als eine Barriere vom Typ-ATOMIC ist. Dies ist aber wahrscheinlich darauf zurückzuführen, dass eine Geforce 280 GTX im Vergleich zu modernen GPUs noch keinen L2-Cache besitzt, weshalb die GPU sämtliche atomaren Operationen direkt auf den DRAM ausführen muss.

10.3.1.5 Wahl des Mappings zwischen den Workitems und Vektorkomponenten

Während ein Tile für sämtliche seiner Vektorgrundoperation für die Datenwiederverwendung über den Registersatz das gleiche Mapping zwischen Vektorgrundoperationen und Workitems besitzen muss, können wir das Mapping ansonsten frei wählen. Dieses Mapping sollte ausgehend von den 4-, 8- und 16-Byte-Gather-Instruktionen sowie den 4-, 8- und 16-Byte-Scatter-

Instruktionen, die eine moderne GPU typischerweise unterstützt, ein gutes Coalescing und eine gute Vektorisierung für das jeweilige Anfangswertproblem ermöglichen.

Für viele Anfangswertprobleme wie Stencils lässt sich im Falle von einfacher Gleitkommagenauigkeit ein solches gutes Coalescing erzielen, indem ein Thread beziehungsweise dessen Workitems einen oder mehrere Blöcke von aufeinanderfolgenden Vektorkomponenten, welche jeweils ein-, zwei- oder viermal die SIMD-Breite groß sind, berechnet. Analog lässt sich im Falle von doppelter Gleitkommagenauigkeit ein solches gutes Coalescing erzielen, indem ein Thread beziehungsweise dessen Workitems einen oder mehrere Blöcke von aufeinanderfolgenden Vektorkomponenten, welche jeweils ein-, zwei- oder verimal die SIMD-Breite groß sind, berechnet.

Indem wir einem Thread mehrere dieser Blöcke mit einem gegebenen Stride zuweisen, können wir dadurch die Lastbalancierung und die Lokalität optimieren sowie beliebig große Tile-Breiten realisieren. Es gibt mehrere Möglichkeiten, um den Stride zwischen den Blöcken eines Threads zu wählen. Dabei besitzen größere Strides eine geringere Lokalität, aber dafür eine bessere Lastbalancierung. Wir implementierten für das Single-Workgroup-Tiling und für das Multi-Workgroup-Tiling die folgenden drei Strides, die noch einmal in Abbildung Abbildung 10.12 gezeigt werden:

- **Stride-THRD:** Beim *Stride-THRD* umspannt der Stride zwischen den Blöcken eines Threads nur den entsprechenden Block (beste Lokalität, schlechteste Lastbalancierung, sowohl beim Single-Workgroup-Tiling als auch beim Multi-Workgroup-Tiling möglich)
- **Stride-WKGP:** Beim *Stride-WKGP* umspannt der Stride zwischen den Blöcken eines Threads die Threads der Workgroup (mittelmäßige Lokalität, mittelmäßige Lastbalancierung, sowohl beim Single-Workgroup-Tiling als auch beim Multi-Workgroup-Tiling möglich)
- **Stride-TILE:** Beim *Stride-TILE* umspannt der Stride zwischen den Blöcken eines Threads die Workgroups des Tiles (schlechteste Lokalität, beste Lastbalancierung, offensichtlich nur beim Multi-Workgroup-Tiling möglich)

10.3.1.6 Deadlock-Freiheit für das Multi-Workgroup-Tiling

Da unsere Implementierung des Multi-Workgroup-Tilings jeweils mehrere aufeinanderfolgende Workgroups einem Tile zuweist, setzt sie, um ein Deadlock zwischen den Workgroups zu vermeiden, zusätzlich Folgendes voraus:

- Der Task-Scheduler der GPU weist die Workgroups in der bezüglich ihres Indexes aufsteigenden Reihenfolge den Kernen der GPU zu.

- Der Task-Scheduler der GPU weist eine Workgroup nur einem Kern der GPU zu, wenn dieser aktuell genügend freie Ressourcen besitzt, um umgehend mit der Ausführung dieser Workgroup zu beginnen.

Bedauerlicherweise garantieren die GPU-Hersteller weder, dass ihre GPUs die Workgroups eines Kernels gleichzeitig ausführen, noch, dass ihre GPUs die Workgroups eines Kernels in einer bestimmten Reihenfolge ausführen. Jedoch funktionierte unsere Implementierung des Multi-Workgroup-Tilings problemlos auf sämtlichen betrachteten AMD- und NVIDIA-GPUs,

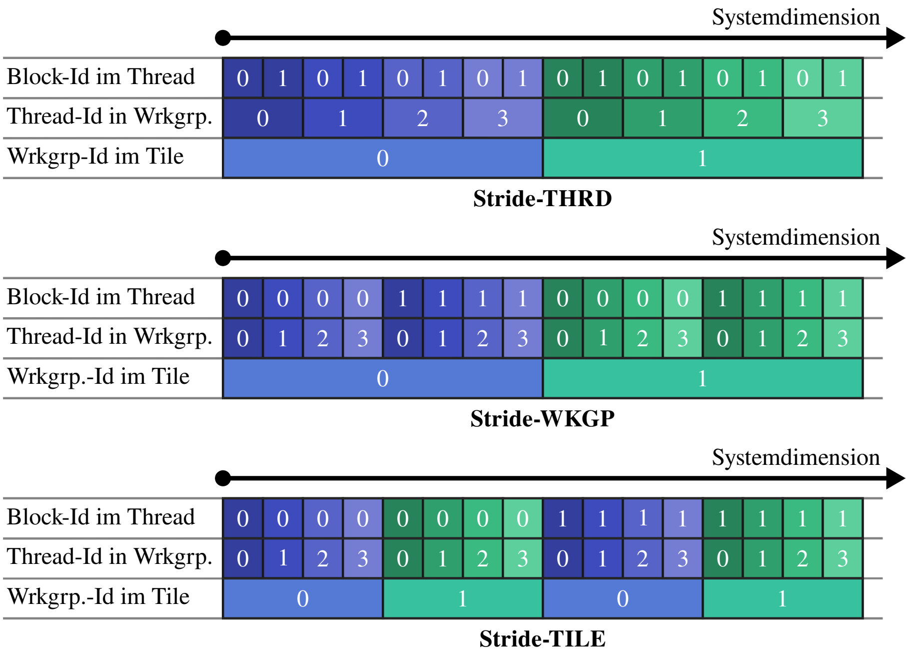


Abbildung 10.12: Implementierte Strides beim Tiling auf GPUs. Diese Abbildung zeigt die drei implementierten Strides zwischen den von einem Thread berechneten Blöcken. Dabei umspannt Stride-THRD den Block eines Threads, Stride-WKGP die Threads einer Workgroup und Stride-TILE die Workgroups eines Tiles.

weshalb auf diesen GPUs die soeben genannten Voraussetzungen für das Multi-Workgroup-Tiling sehr wahrscheinlich erfüllt sind. Zusätzlich werden diese Voraussetzungen nicht bei einer Implementierung des Multi-Workgroup-Tilings benötigt, die auf einem *Persistent-Threading-Ansatz* basiert. Eine solche Implementierung des Multi-Workgroup-Tilings mit dem Persistent-Threading-Ansatz startet nur so viele Workgroups, dass die GPU komplett belegt ist, und lässt jede Gruppe von Workgroups nacheinander mehrere Multi-Workgroup-Tiles berechnen.

10.3.1.7 Überlegungen zur Tile-Breite und zum Autotuning

Während die Tile-Höhe ein unabhängiger Parameter für das Tiling ist, so wird die Tile-Breite durch mehrere interagierende Parameter bestimmt:

- n_{comp}: Die Anzahl an Komponenten per Block
- n_{blocks}: Die Anzahl an Blöcken per Thread
- n_{threads}: Die Anzahl an Threads per Workgroup
- n_{wgtile}: Die Anzahl an Workgroups per Tile
- n_{wgcore}: Die Anzahl an simultanen Workgroups per Kern

- n_{tiles}: Die Anzahl an simultanen Tiles auf der GPU

Wir bezeichnen diese Menge an Parametern als eine *Tile-Breitenkonfiguration*, die in einer Tile-Breite von w resultiert:

$$w = n_{\text{comp}} \cdot n_{\text{blocks}} \cdot n_{\text{threads}} \cdot n_{\text{wgtile}}. \tag{10.17}$$

Offensichtlich ist bei einer Tile-Breitenkonfiguration für das Single-Workgroup-Tiling $n_{\text{wgtile}} = 1$, während bei einer Tile-Breitenkonfiguration für das Multi-Workgroup-Tiling $n_{\text{wgtile}} > 1$ ist.

Um eine gute Performance zu erzielen, müssen wir alle Parameter der Tile-Breitenkonfiguration passend wählen. Zuerst wählen wir die Anzahl an Blöcken per Thread, wofür ein kleiner Wert von 1, 2, 3 oder 4 typischerweise die beste Performance liefert. Als Nächstes wählen wir die Anzahl an Threads per Workgroup (n_{threads}) und die Anzahl an simultanen Workgroups per Kern (n_{wgcore}) wie folgt:

- Für die Anzahl an nebenläufige Workgroups per Kern wird ein möglichst kleiner Wert gewählt, um den Overhead der Multi-Workgroup-Barrieren zu reduzieren.
- Die Anzahl an nebenläufige Threads per Workgroup beziehungsweise die Anzahl an nebenläufige Threads per Kern verursacht kein oder nur wenig Registerspilling.
- Ein Kern kann die maximale Anzahl an Threads für die gegebene Registerallozierung des Kernels beherbergen.

In Abhängigkeit von den Occupancy-Regeln der GPU gibt es typischerweise nur wenige vielversprechende Werte für beides.

Zuletzt müssen wir die Anzahl an Workgroups pro Tile (n_{wgtile}) wählen, welche indirekt die Anzahl an simultan von der GPU ausgeführten Tiles (n_{tiles}) bestimmt. Um die Ressourcenauslastung auf der GPU zu maximieren, sollten die gewählten Werte für n_{wgtile} und n_{tiles} folgende Gleichungen erfüllen:

$$\left\lfloor \frac{n_{\text{wgcore}} \cdot n_{\text{cores}}}{n_{\text{wgtile}}} \right\rfloor = n_{\text{tiles}} \quad \text{und} \quad \left\lfloor \frac{n_{\text{wgcore}} \cdot n_{\text{cores}}}{n_{\text{tiles}}} \right\rfloor = n_{\text{wgtile}}, \tag{10.18}$$

wobei n_{cores} die Anzahl an Kerne auf der GPU ist.

Insgesamt ergeben sich so nur wenige vielversprechende Tile-Breiten, welche unser Autotuning-Ansatz aus Abschnitt 10.2.11 zur Bestimmung der optimalen Konfiguration des Tilings leicht für jedes Kernel evaluieren kann. Alle anderen Parameter der Tile-Breitenkonfiguration muss dieser Autotuning-Ansatz als Bestandteil der Kernelkonfiguration interpretieren. Diese Kernelkonfiguration kann für GPUs zusätzlich noch weitere Parameter, wie den Stride, die verwendete Barriere oder die Art des Bufferings, beinhalten. Somit muss unser Autotuning-Ansatz für jede vielversprechende Tile-Breite eine vollständige Suche über sämtliche Tile-Breitenkonfigurationen, welche ebendiese Tile-Breite ergeben, und zusätzlich noch über sämtliche anderen Parameter der Kernelkonfiguration durchführen, um dadurch die performanteste Tile-Breitenkonfiguration beziehungsweise Kernelkonfiguration für diese Tile-Breite zu ermitteln.

10.3.1.8 Codegenerierung

Wir implementierten einen Codegenerator, welcher ausgehend von einem Datenflussgraphen und einem Tiling automatisch für GPUs den Quelltext der Kernels in CUDA-C oder OpenCL-C erzeugt. Diesen Quelltext lassen wir dann von der CUDA-API beziehungsweise von der OpenCL-API zu einem GPU-Kernel kompilieren, welches wir dann mit der jeweiligen API auch ausführen. Ein schematisches Beispiel für diesen generierten Code wird von Abbildung 10.13 gezeigt.

Für ein Kernel mit Tiling erzeugt unser Codegenerator einen Quelltext mit sämtlichen Speicheroptimierungen, wie wir sie in diesem Kapitel beschrieben. Insbesondere gilt wieder, dass wir die Datenwiederverwendung über den Registersatz und den Datentransfer über den selbigen mit Hilfe von automatischen Variablen erzielen, welche ein GPU-Compiler immer in Registern der GPU platziert. Des Weiteren entfernt unser Codegenerator bereits redundante Ladeoperationen aus dem generierten Quelltext, indem er die vom Tile benötigten Komponenten eines permanenten Vektors zunächst in einer automatischen Variable abspeichert, und diese automatische Variable dann als Argument für sämtliche fusionierten (LC | MAP | RED)-Operationen, die auf diesen Vektor lesend zugreifen, verwendet.

Bei der Codegenerierung fügt der Codegenerator die Vektorgrundoperationen in ihrer topologischen Sortierung in den Quelltext des Kernels ein. Zusätzlich fügt er zwischen jedem RHS $\rightarrow$ LC-Glied eine Tile umspannende Barriere in den Quelltext des Kernels ein. Soll ein jedes Workitem beziehungsweise ein jeder Thread in einem Tile nicht nur einen Block, sondern mehrere Blöcke von Komponenten berechnen, so realisiert dies unser Codegenerator nicht durch eine Schleife über diese Blöcke, sondern er fügt für jede Vektorgrundoperation und für jeden Block eine Folge von Statements ein:

1. **Statements zum Laden:** Die ersten Statements laden, nur falls nötig, die benötigten Komponenten der permanenten Argumentvektoren der auszuwertenden Vektorgrundoperation aus dem DRAM in eine automatische Variable.
2. **Statements zum Berechnen:** Die nächsten Statements berechnen die Vektorgrundoperation für den entsprechenden Block des entsprechenden Workitems und speichern das Ergebnis der Vektorgrundoperation in einer automatischen Variable ab.
3. **Statements zum Zurückschreiben:** Die letzten Statements schreiben, nur falls nötig, das Ergebnis der Vektorgrundoperation von der automatischen Variable zurück in den dazugehörigen permanenten Vektor und damit in den DRAM oder im Falle einer verschmolzenen LC $\rightarrow$ RHS-Abhängigkeit zusätzlich in einen LC $\rightarrow$ RHS-Buffer.

Um zu erzielen, dass nicht alle Blöcke von den Workitems eines Tiles an dem Laden, dem Berechnen oder an dem Zurückschreiben teilnehmen, versieht unser Codegenerator jedes dieser Statements mit einem zusätzlichen If-Statement, das überprüft, ob für den entsprechenden Block eines Workitem das entsprechende Statement auszuführen ist, also ob dieses Workitem die Komponenten des Blocks aus dem DRAM laden, berechnen oder in den DRAM zurückschreiben soll. Dieses If-Statement kann jedoch entfallen, wenn die zu ladenden, die zu berechnenden oder die zurück zuschreibenden Komponenten die komplette Breite des Tiles umspannen.

```
in: double2* y_k;
out: double2* Y_1, Y_2;

code:
parallel for each workgroup wg
   parallel for each workitem wi in wg
      double2 y_k_b1_reg, y_k_b2_reg, F_1_b1_reg, F_1_b2_reg,
         Y_1_b2_reg, Y_1_b2_reg, F_2_b1_reg, F_2_b2_reg;

      tile t = get_tile(wg);
      int i_b1  = compute_index_of_block_1(t, wi, stride);
      int i_b2  = compute_index_of_block_2(t, wi, stride);
      double2* lc_rhs_buffer = get_lc_rhs_buffer(t);

      first link l1 of t:

         y_k_b1_reg =  y_k[i_b1];
         F_1_b1_reg = RHS(y_k, i_b1);
         Y_1_b2_reg = y_k_b2_reg + a_21 * F_1_b2_reg;
         if i_b1 in left or right border of Y_1 in t
            Y_1[i_b1] = Y_1_b1_reg;
         lc_rhs_buffer[i_b1] = Y_1_b1_reg;

         y_k_b2_reg =  y_k[i_b2];
         F_1_b2_reg = RHS(y_k, i_b2);
         Y_1_b2_reg = y_k_b2_reg + a_21 * F_1_b2_reg;
         if i_b2 in left or right border of Y_1 in t
            Y_1[i_b2] = Y_1_b2_reg;
         lc_rhs_buffer[i_b2] = Y_1_b2_reg;

      tile_barrier(t);

      second link l2 of t:

         if i_b1 in argument range of l2
            F_2_b1_reg = RHS(lc_rhs_buffer, i_b1);
            Y_2_b1_reg = y_k_b1_reg + a_31 * F_1_b1_reg +
               a_32 * F_1_b1_reg;
            Y_2[i_b1] = Y_2_b1_reg;

         if i_b2 in argument range of l2
            F_2_b2_reg = RHS(lc_rhs_buffer, i_b2);
            Y_2_b2_reg = y_k_b2_reg + a_31 * F_1_b2_reg +
               a_32 * F_1_b2_reg;
            Y_2[i_b2] = Y_2_b2_reg;
```

Abbildung 10.13: Generierter Code für ein GPU-Kernel mit Tiling. Dabei zeigt dieses Beispiel den Quelltext für ein GPU-Kernel, das ein schrumpfendes Trapezoid berechnet.

10.3.2 Experimente

10.3.2.1 Aufbau der Experimente

Betrachtete ODE-Verfahren: Als ODE-Verfahren verwenden wir, um das Tiling über die Stufen zu untersuchen, zwei unterschiedliche explizite eingebettete RK-Verfahren mit unterschiedlicher Ordnung: das Bogacki–Shampine-2(3)-Verfahren mit vier Stufen und das Verner-5(6)-Verfahren mit acht Stufen. Um zusätzlich das Tiling über die Zeitschritte zu untersuchen, verwenden wir folgende RK-Verfahren: das explizite Euler-Verfahren mit einer Stufe, das Heun-Verfahren mit zwei Stufen und das RK-3/8-Verfahren mit vier Stufen. Die Koeffizienten dieser RK-Verfahren werden im Abschnitt 13.7 gezeigt.

Testproblem und Problemgröße: Für sämtliche Messungen mit der beschränkten Zugriffsdistanz verwenden wir BRUSS2D als zu lösendes Anfangswertproblem. Da es sich bei BRUSS2D um einen 5-Punkt-Stencil auf einem 2D-Gitter mit zwei Werten pro Gitterzelle handelt, die wir in der Row-Major-Anordnung als Array-of-Structures abspeichern, besitzt es bei einer Kantenlänge von xmax Gitterzellen entlang der x-Achse eine beschränkte Zugriffsdistanz d von $d = 2 \cdot \text{xmax}$. Um den Einfluss der beschränkten Zugriffsdistanz auf die Performance zu untersuchen, setzen wir Anzahl der Gitterzellen entlang der x-Achse auf $\text{xmax} = d/2$ Gitterzellen und passen Anzahl der Gitterzellen entlang der y-Achse so an, dass die Größe des ODE-Systems n insgesamt ungefähr konstant bleibt.

Für diese Problemgröße n wählten wir für die Messungen mit doppelter Genauigkeit einen Wert von $n = 32 \cdot 1024 \cdot 1024$ Komponenten und für die Messungen mit einfacher Genauigkeit einen Wert von $n = 64 \cdot 1024 \cdot 1024$ Komponenten. Eine Ausnahme hierfür stellen die Messreihen, welche das Tiling über die Zeitschritte untersuchen, dar, für welche wir aus historischen Gründen eine Problemgröße von $n = 4 \cdot 1024 \cdot 1024$ wählten. Gemäß der Problemstellung nutzen wir nur die beschränkte Zugriffsdistanz von BRUSS2D aus. Somit spannt BRUSS2D über die Systemdimension und die Zeitdimension einen zweidimensionalen Iterationsraum auf, den wir für unser Tiling verwenden können. Jedoch nutzen wir nicht die zweidimensionale räumliche Struktur des Stencils aus, welche für BRUSS2D einen dreidimensionalen Iterationsraum ergeben würde (x-, y- und Zeitdimension).

Referenzimplementierung: Als Referenzimplementierung für die Berechnung des Speedups verwenden wir für das jeweilige ODE-Verfahren eine Implementierungsvariante nur mit Kernelfusion, aber ohne Tiling. Diese Referenzimplementierung verschmilzt somit nur (RHS | LC | MAP) $\rightarrow$ (LC | MAP | RED)-Abhängigkeiten, aber keine LC $\rightarrow$ RHS-Abhängigkeiten. Dadurch ist die Referenzimplementierung ähnlich zu den Varianten vom Typ-FUS aus dem Abschnitt 9.2.8. Jedoch startet unsere Referenzimplementierung wegen der Problemstellung in diesem Kapitel nur 1D-Kernels. Dadurch nutzt sie im Vergleich zu den Varianten mit Kernelfusion nicht durch 2D-Kernels die 2D-Struktur des Stencils zur Erhöhung der räumlichen Lokalität der Speicherzugriffe aus. Ebenso verwendet unsere Referenzimplementierung aus historischen Gründen keine Elbling-Transformationen.

Bestimmung der optimalen Tile-Höhen und der optimalen Tile-Breitenkonfigurationen: Hierfür verwenden wir den im Abschnitt 10.2.11 und im Abschnitt 10.3.1.7 beschriebenen Ansatz, mit welchem wir für jedes Tiling-Schema die optimalen Tile-Breiten, die optimalen Tile-Höhen und zusätzlich für jedes Kernel die optimalen Tile-Breitenkonfigurationen beziehungsweise Kernelkonfigurationen ermitteln.

Verwendete GPUs und GPGPU-APIs: Um möglichst viele Architekturen abzudecken, ver-

wenden wir drei unterschiedliche GPUs als Plattformen für unsere Experimente:

- **Titan Volta (*Volta*):** Messung der Laufzeiten und der Speedups in Abhängigkeit von der Zugriffsdistanz für das Verner-Verfahren und das Bogacki–Shampine-Verfahren mit doppelter Gleitkommagenauigkeit. Zudem Untersuchungen zur Tile-Breite, Tile-Höhe und Profiling für das Verner-Verfahren.
- **Geforce Titan Black (*Kepler*):** Messung der Laufzeiten und der Speedups in Abhängigkeit von der Zugriffsdistanz für das Verner-Verfahren mit doppelter Gleitkommagenauigkeit. Zudem Untersuchungen für das Tiling über die Zeitschritte mit dem Euler-Verfahren, dem Heun-Verfahren und dem RK-3/8-Verfahren. Zusätzliches Profiling für das Verner-Verfahren.
- **Radeon RX 480 (*Polaris*):** Messung der Laufzeit und des Speedups in Abhängigkeit von der Zugriffsdistanz für das Verner-Verfahren mit einfacher Gleitkommagenauigkeit.

Dabei verwendeten wir für die Volta- und Kepler-GPU die CUDA-API, während wir für die Polaris-GPU die OpenCL-API verwendeten. Da sowohl die Volta-GPU als auch die Kepler-GPU eine gute DP-Performance besitzen, führen wir auf beiden GPUs sämtliche Messungen mit doppelter Gleitkommagenauigkeit durch. Dahingegen besitzt die Polaris-GPU nur eine sehr schlechte DP-Performance, weshalb wir auf ihr sämtliche Messungen nur mit einfacher Gleitkommagenauigkeit durchführen.

10.3.2.2 Messungen

Mit diesem Aufbau führten wir folgende Messreihen durch:

- **Untersuchungen zur Laufzeit und zum Speedup für das Tiling über die Stufen (Abbildung 10.14, Abbildung 10.15, Abbildung 10.16 und Abbildung 10.17):** Einfluss der Zugriffsdistanz auf die inverse Laufzeit und den Speedup für die beiden Tiling-Schemata mit den optimalen Tile-Breitenkonfigurationen und den optimalen Tile-Höhen für das Verner-Verfahren auf Volta, für das Bogacki-Shampine-Verfahren auf Volta, für das Verner-Verfahren auf Kepler und für das Verner-Verfahren auf Polaris.
- **Untersuchungen zur Tile-Breite und der Anzahl an nebenläufigen Tiles (Abbildung 10.18):** Die beste Tile-Breite und die beste Anzahl an nebenläufigen Tiles in Abhängigkeit von der Zugriffsdistanz für mehrere fest gewählte Tile-Höhen, das hexagonale Multi-Workgroup-Tiling und das Verner-Verfahren auf Volta.
- **Untersuchungen zu den Blöcken pro Thread und den Strides (Abbildung 10.19):** Einfluss der Blöcke pro Thread und der Strides auf die Performance für das Verner-Verfahren und das hexagonale Multi-Workgroup-Tiling für eine Tile-Höhe von vier auf Volta.
- **Untersuchungen zu den Tile-Höhen (Abbildung 10.20):** Einfluss von unterschiedlichen Tile-Höhen mit der optimalen Tile-Breitenkonfiguration auf den Speedup des Multi-Workgroup-Tilings in Abhängigkeit von der Zugriffsdistanz für das Verner-Verfahren auf Volta.
- **Profiling (Abbildung 10.21 und Abbildung 10.22):** Profiling der Auslastung der DP-Einheiten (U_{DP}), der Auslastung der IPC (U_{IPC}), der Auslastung der DRAM-Bandbreite (U_{DRAM}) und des DRAM-Volumens (V_{DRAM}) auf Volta und auf Kepler. Eine genaue Beschreibung dieser Metriken ist im Abschnitt 13.4 zu finden.

- **Untersuchungen zur Laufzeit und zum Speedup für das Tiling über die Zeitschritte (Abbildung 10.23, Abbildung 10.24 und Abbildung 10.25):** Einfluss der Zugriffsdistanz auf die inverse Laufzeit und den Speedup für die beiden Tiling-Schemata mit den optimalen Tile-Breitenkonfigurationen und den optimalen Tile-Höhen für das explizite Euler-Verfahren, das Heun-Verfahren und das Rk-3/8-Verfahren auf Kepler.

Zu den Messungen ist Folgendes anzumerken:

- Wir wählten für die Achse der Zugriffsdistanz in den Diagrammen mit dem Multi-Workgroup-Tiling immer eine logarithmische Skalierung, damit wir das Verhalten des Tilings für einen großen Bereich an Zugriffsdistanzen sinnvoll visualisieren können. Diese logarithmische Skalierung erweckt jedoch auf den ersten Blick den visuellen Eindruck, dass das Single-Workgroup-Tiling für einen großen Bereich an Zugriffsdistanzen mit dem Multi-Workgroup-Tiling konkurrieren kann. In Wirklichkeit lohnt sich aber zum Beispiel auf Volta das Single-Workgroup-Tiling gegenüber dem Multi-Workgroup-Tiling nur für Zugriffsdistanzen, die geringer als ungefähr $2.5 \cdot 10^2$ sind, während das Multi-Workgroup-Tiling noch für Zugriffsdistanzen von ungefähr $2.0 \cdot 10^5$ schneller als die Referenzimplementierung ist.
- Aus diesem Grund verzichten wir in dieser Arbeit auf eine ausgiebige Untersuchung des Single-Workgroup-Tilings. Eine solche ausgiebige Untersuchung des Single-Workgroup-Tilings ist in unserer Arbeit [E4] zu finden.
- Beim Multi-Workgroup-Tiling traten auf Polaris Probleme auf, welche das Multi-Workgroup-Tiling weniger effizient machten. Diese Probleme werden wir in dem Abschnitt 13.3.2 erläutern.
- Leider funktioniert aktuell unser Autotuning für das Multi-Workgroup-Tiling nur für das Tiling über die Stufen eines Zeitschritts, nicht aber für das Tiling über die Zeitschritte. Deshalb konnten wir das Tiling über die Zeitschritte nur mit dem Single-Workgroup-Tiling untersuchen.
- Offensichtlich ist beim expliziten Euler-Verfahren, da es nur eine Stufe besitzt, kein Tiling über dessen Stufen, sondern nur ein Tiling über die Zeitschritte möglich.
- Aus historischen Gründen verwendeten wir weder bei den Varianten mit Tiling noch bei der Referenz-Implementierung die Enabling-Transformationen aus dem Kapitel 9.

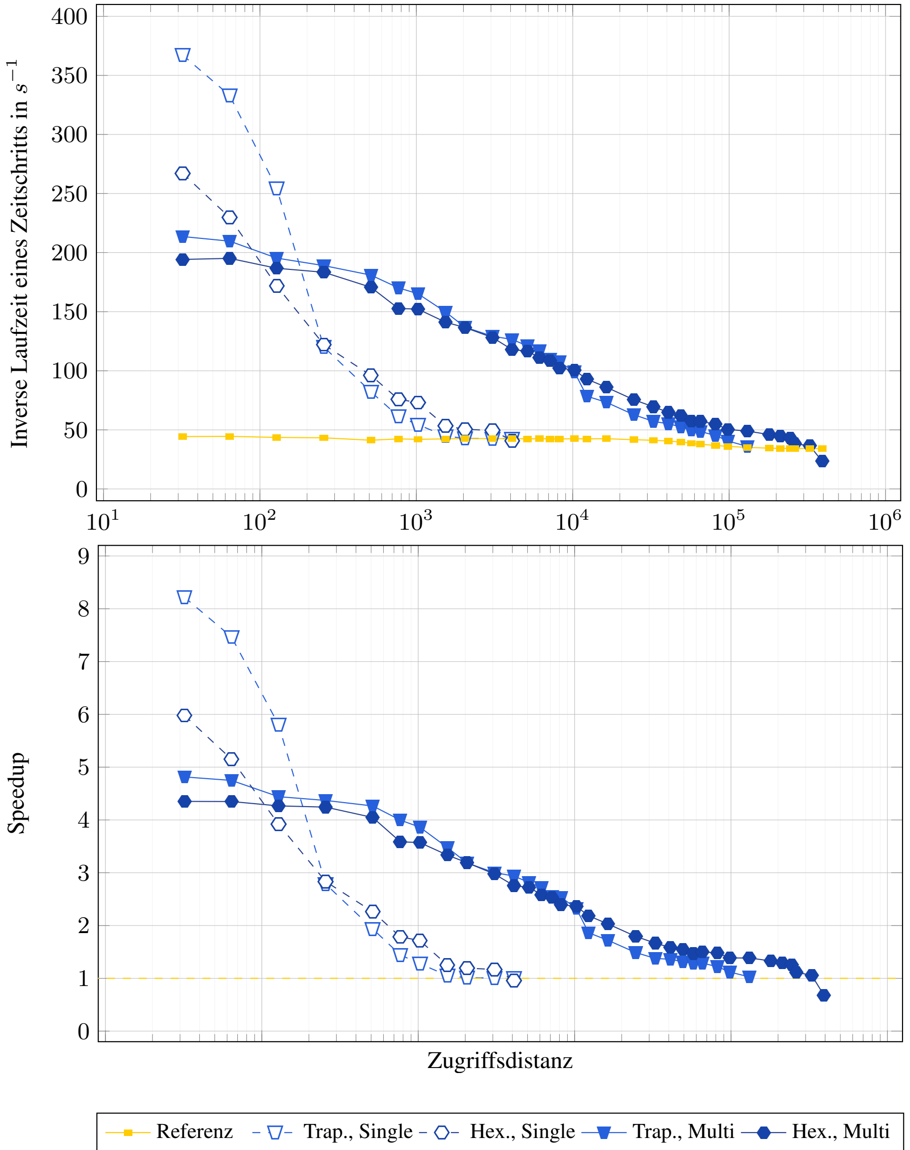


Abbildung 10.14: Laufzeituntersuchungen für das Verner-Verfahren mit Tiling auf Volta. Diese Untersuchungen zeigen den Einfluss der Zugriffsdistanz auf die inverse Laufzeit und den Speedup für die beiden Tiling-Schemata mit den optimalen Tile-Breitenkonfigurationen und den optimalen Tile-Höhen.

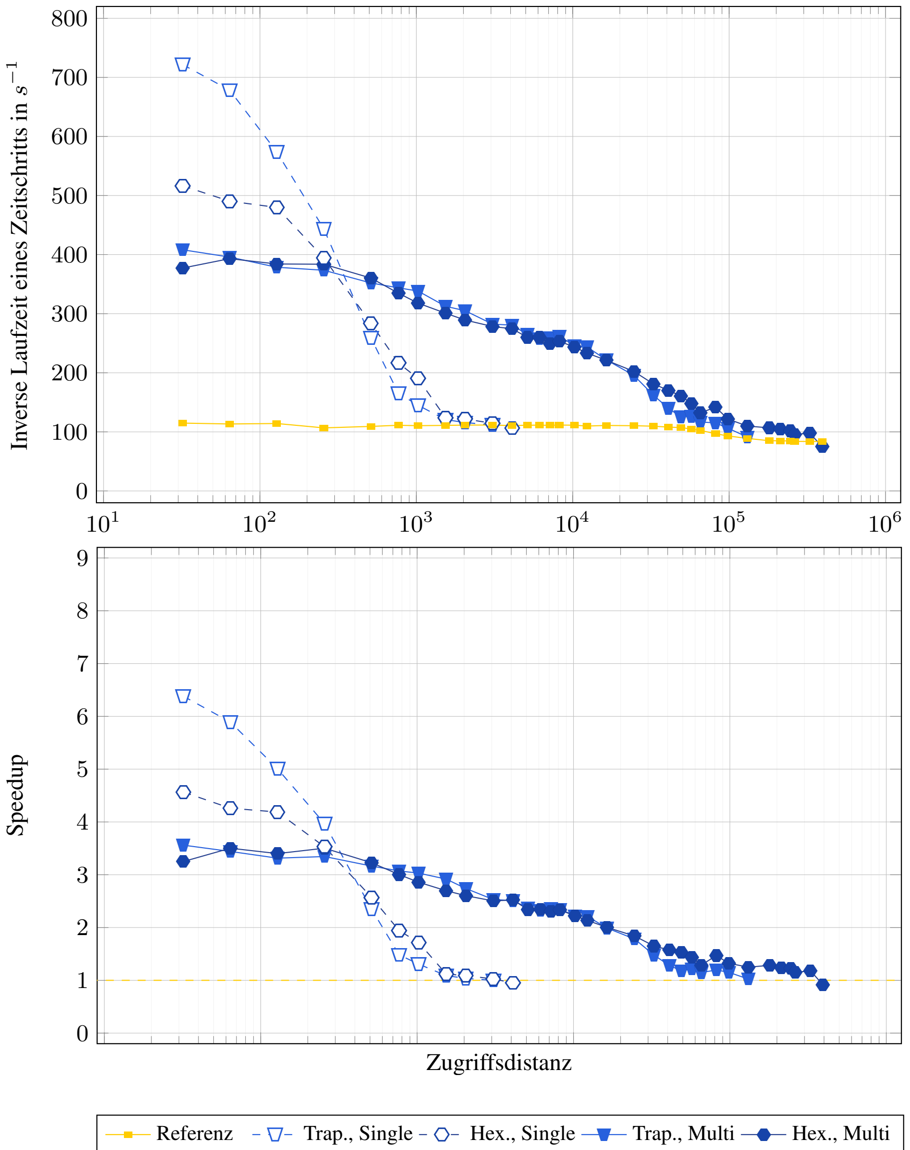


Abbildung 10.15: Laufzeituntersuchungen für das Bogacki-Shampine-Verfahren mit Tiling auf Volta. Diese Untersuchungen zeigen den Einfluss der Zugriffsdistanz auf die inverse Laufzeit und den Speedup für die beiden Tiling-Schemata mit den optimalen Tile-Breitenkonfigurationen und den optimalen Tile-Höhen.

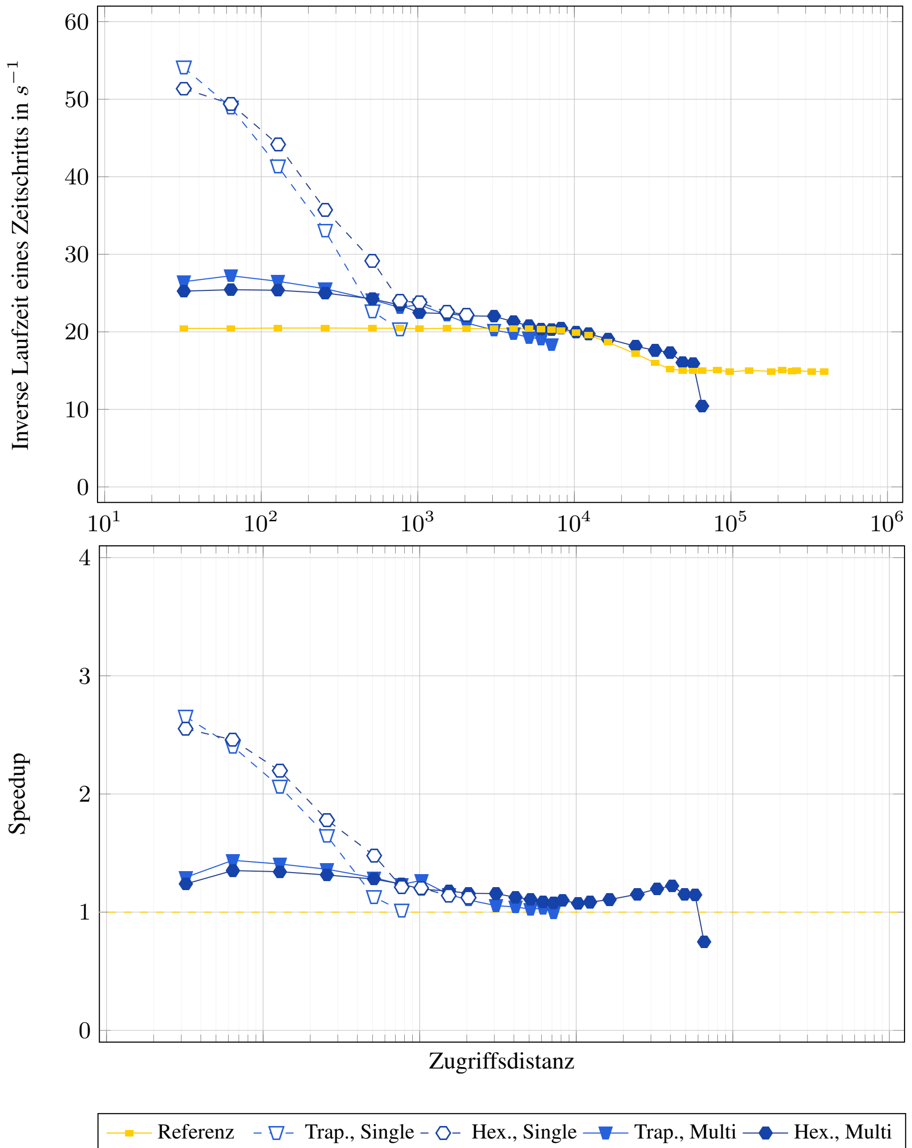


Abbildung 10.16: Laufzeituntersuchungen für das Verner-Verfahren mit Tiling auf Kepler. Diese Untersuchungen zeigen den Einfluss der Zugriffsdistanz auf die inverse Laufzeit und den Speedup für die beiden Tiling-Schemata mit den optimalen Tile-Breitenkonfigurationen und den optimalen Tile-Höhen.

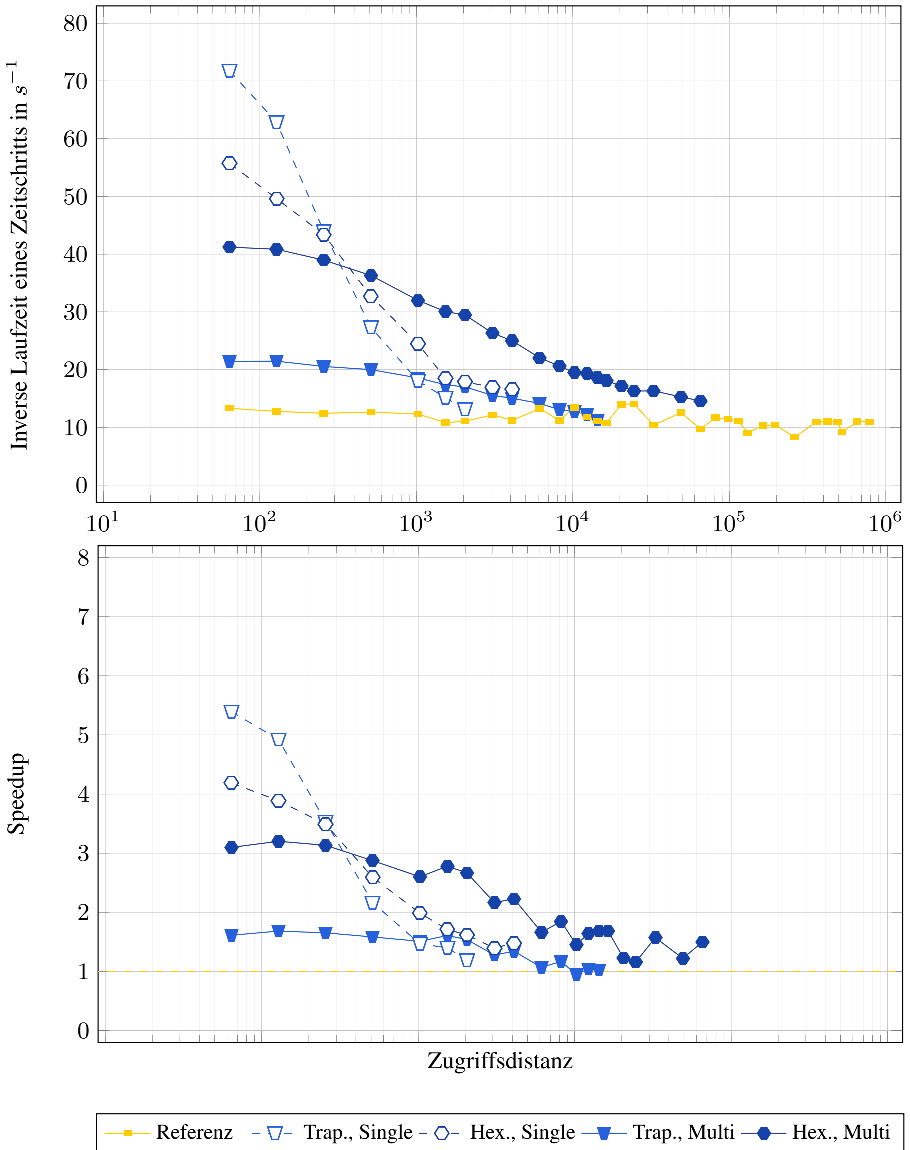


Abbildung 10.17: Laufzeituntersuchungen für das Verner-Verfahren mit Tiling auf Polaris. Diese Untersuchungen zeigen den Einfluss der Zugriffsdistanz auf die inverse Laufzeit und den Speedup für die beiden Tiling-Schemata mit den optimalen Tile-Breitenkonfigurationen und den optimalen Tile-Höhen.

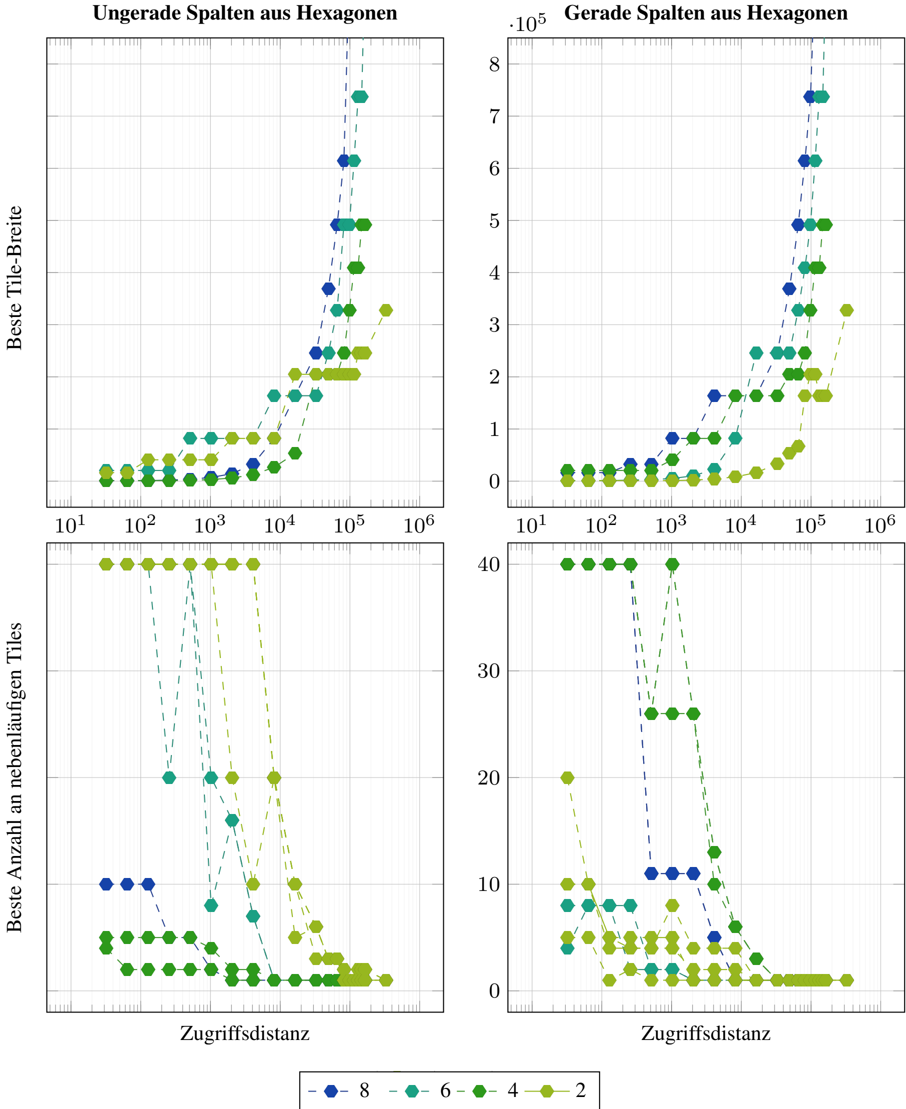


Abbildung 10.18: Untersuchungen zur besten Tile-Breite und der besten Anzahl an nebenläufigen Tiles beim Multi-Workgroup-Tiling. Hierbei untersuchen wir den Einfluss der Zugriffsdistanz auf die beste Tile-Breite und die beste Anzahl an nebenläufigen Tiles für mehrere fest gewählte Tile-Höhen (Legende) für das Verner-Verfahren und das hexagonalen Multi-Workgroup-Tiling auf Volta. Für die beste Anzahl an nebenläufigen Tiles stellt das Diagramm jedes Kernel bei einer bestimmten Tile-Höhe über eine eigene Linie dar.

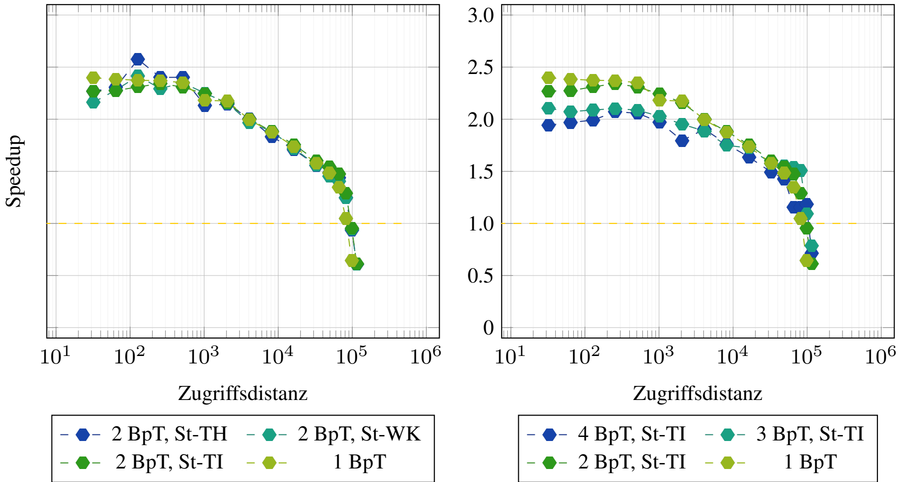


Abbildung 10.19: Einfluss der Blöcke per Thread (BpT) und der Strides (St) beim Multi-Workgroup-Tiling auf die Performance. Die Messungen wurden für das Verner-Verfahren und dem hexagonalen Multi-Workgroup-Tiling mit einer Tile-Höhe von vier auf Volta durchgeführt.

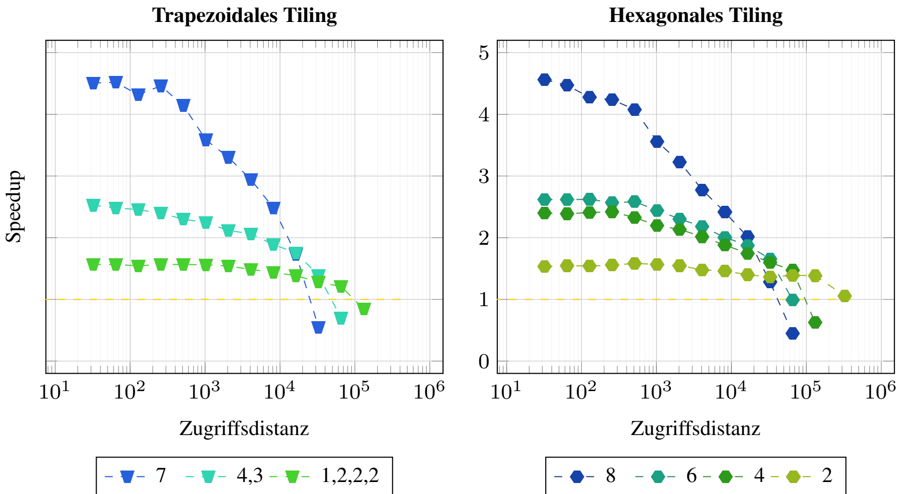


Abbildung 10.20: Untersuchungen zur besten Tile-Höhe beim Multi-Workgroup-Tiling. Diese Messung untersucht den Einfluss der Zugriffsdistanz auf den Speedup der beiden Tiling-Schemata für unterschiedliche Tile-Höhen mit der optimalen Tile-Breitenkonfiguration für das Verner-Verfahren auf Volta. Dabei geben die Einträge in der Legende für das trapezoidale Tiling die Tile-Höhen für die Reihen aus Trapezoiden entlang der Abhängigkeitskette an.

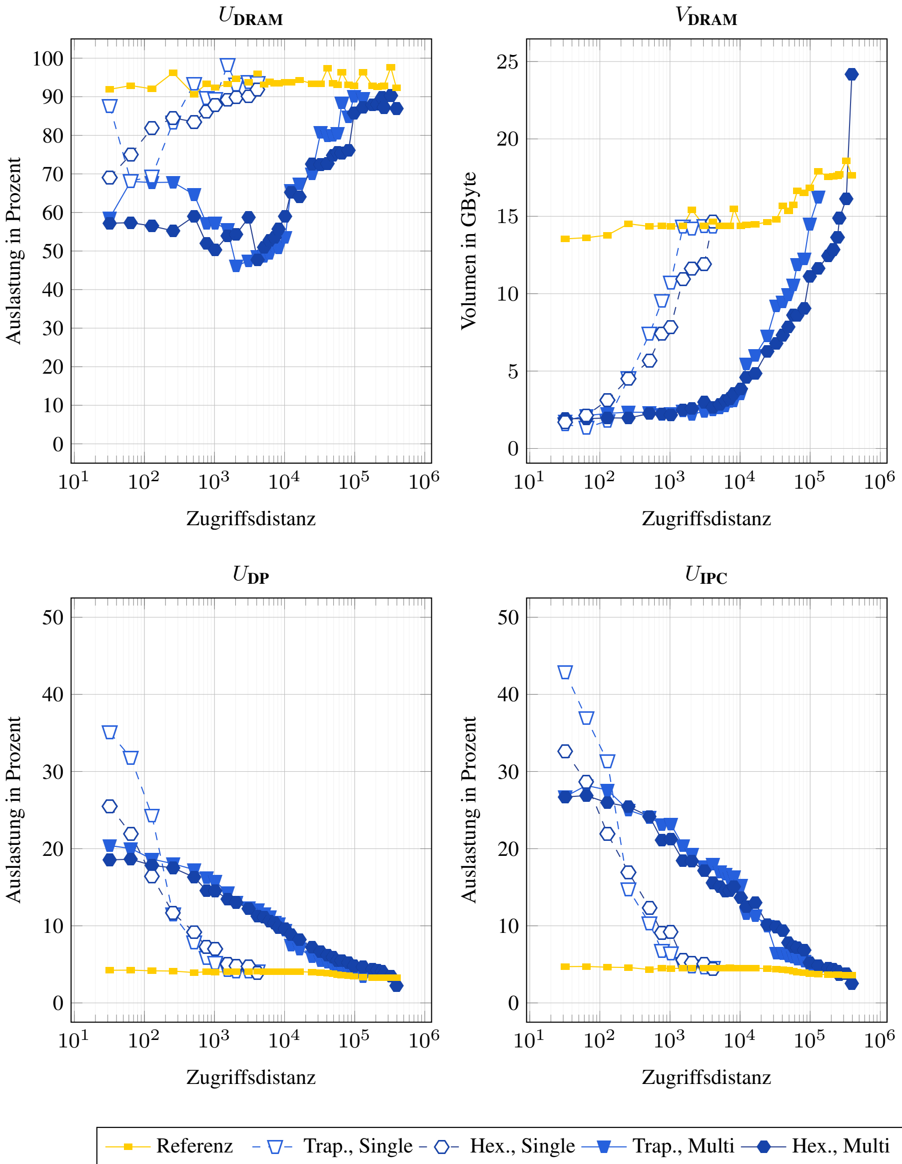


Abbildung 10.21: Profiling des Tilings auf Volta. Dieses Profiling zeigt die Auslastung der DRAM-Bandbreite (U_{DRAM}), das DRAM-Volumen (V_{DRAM}), sowie die Auslastung der DP-Einheiten (U_{DP}) und der IPC (U_{IPC}) für das Verner-Verfahren.

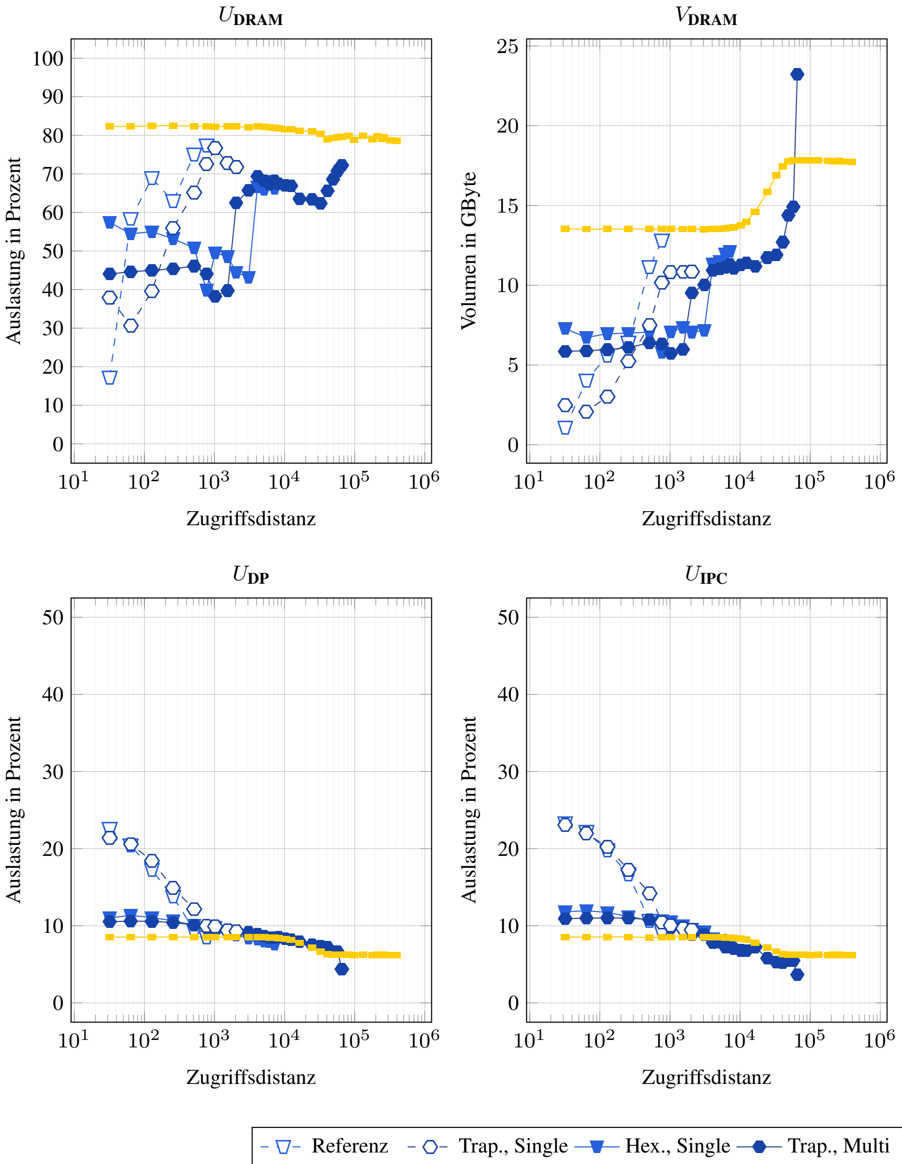


Abbildung 10.22: Profiling des Tilings auf Kepler. Dieses Profiling zeigt die Auslastung der DRAM-Bandbreite (U_{DRAM}), das DRAM-Volumen (V_{DRAM}), sowie die Auslastung der DP-Einheiten (U_{DP}) und der IPC (U_{IPC}) für das Verner-Verfahren.

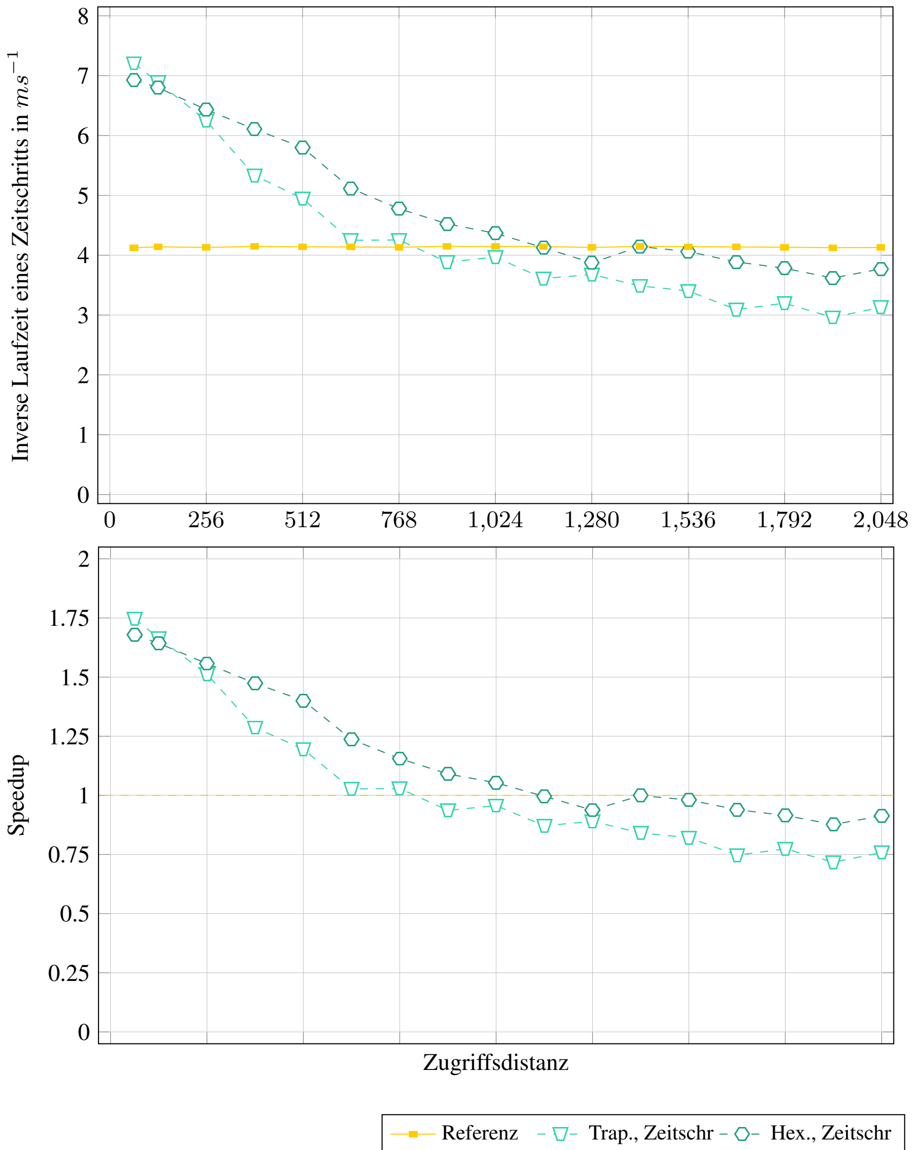


Abbildung 10.23: Laufzeituntersuchungen für das Euler-Verfahren mit Single-Workgroup-Tiling auf Kepler. Diese Untersuchungen zeigen den Einfluss der Zugriffsdistanz auf die inverse Laufzeit und den Speedup für die beiden Tiling-Schemata mit den optimalen Tile-Breitenkonfigurationen und den optimalen Tile-Höhen.

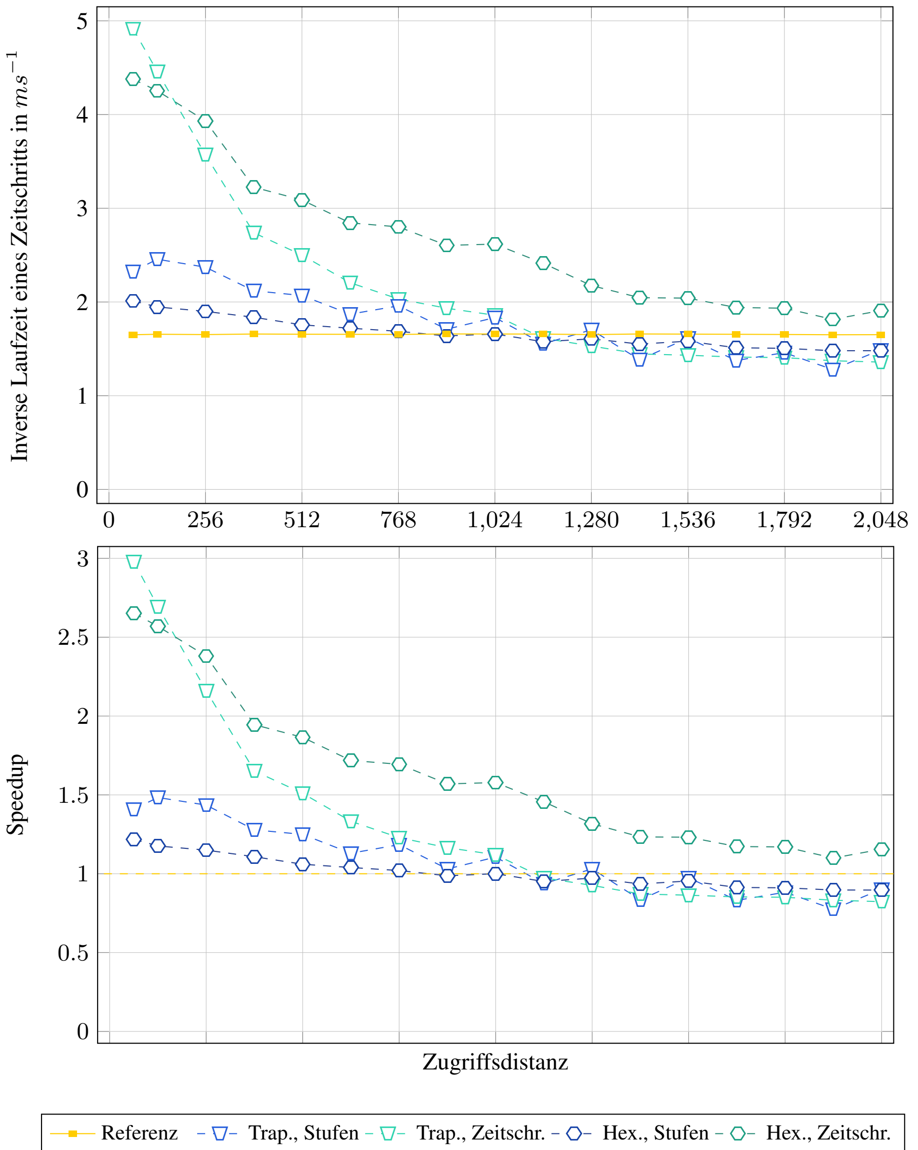


Abbildung 10.24: Laufzeituntersuchungen für das Heun-Verfahren mit Single-Workgroup-Tiling auf Kepler. Diese Untersuchungen zeigen den Einfluss der Zugriffsdistanz auf die inverse Laufzeit und den Speedup für die beiden Tiling-Schemata mit den optimalen Tile-Breitenkonfigurationen und den optimalen Tile-Höhen.

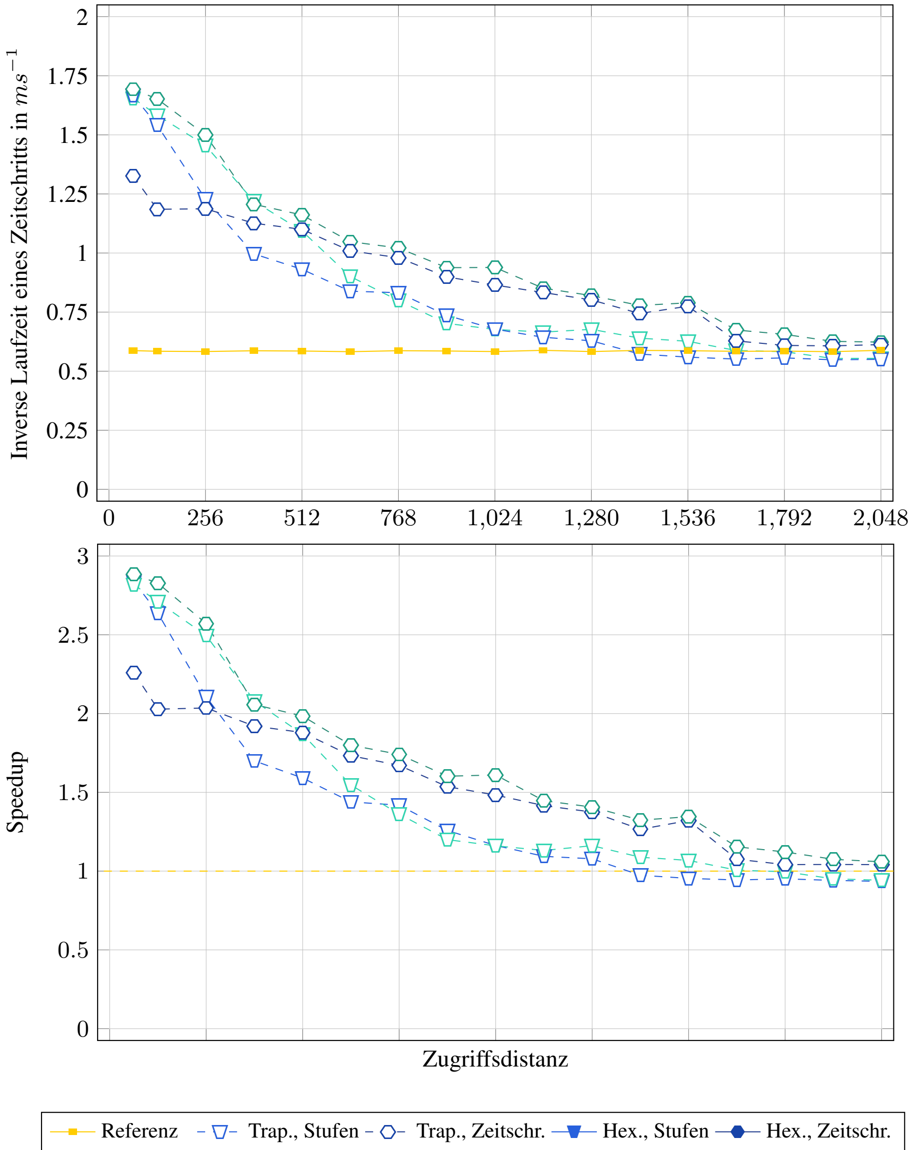


Abbildung 10.25: Laufzeituntersuchungen für das RK-3/8-Verfahren mit Single-Workgroup-Tiling auf Kepler. Diese Untersuchungen zeigen den Einfluss der Zugriffsdistanz auf die inverse Laufzeit und den Speedup für die beiden Tiling-Schemata mit den optimalen Tile-Breitenkonfigurationen und den optimalen Tile-Höhen.

10.3.2.3 Diskussion

Mit Hilfe der Messungen können wir folgende Fragen beantworten:

Für welche Zugriffsdistanzen lohnt sich das Single-Workgroup-Tiling?
(Siehe Abbildung 10.14, Abbildung 10.15, Abbildung 10.16 und Abbildung 10.17)

Auf allen betrachteten Architekturen und allen betrachteten Verfahren lässt sich mit dem Single-Workgroup-Tiling bei sehr kleinen Zugriffsdistanzen ein großer Speedup gegenüber der jeweiligen Referenzimplementierung nur mit Kernelfusion, aber ohne Tiling erzielen, wie zum Beispiel bei einer beschränkten Zugriffsdistanz von 32 ein Speedup von 8.2 für das Verner-Verfahren auf Volta oder ein Speedup von 2.7 für das Verner-Verfahren auf Kepler. Selbst wenn das Single-Workgroup-Tiling für sehr kleine Zugriffsdistanzen einen deutlich größeren Speedup als das Multi-Workgroup-Tiling erzielen kann, so skaliert es deutlich schlechter. Somit ist das Single-Workgroup-Tiling zum Beispiel auf Volta nur bis zu einer Zugriffsdistanz von 256 schneller als das Multi-Workgroup-Tiling und bis zu einer Zugriffsdistanz von ungefähr 1 536 zwar langsamer als das Multi-Workgroup-Tiling, aber immer noch schneller als die Referenzimplementierung.

Für welche Zugriffsdistanzen lohnt sich das Multi-Workgroup-Tiling?
(Siehe Abbildung 10.14, Abbildung 10.15, Abbildung 10.16 und Abbildung 10.17)

Auf Volta gibt es einen großen Bereich an Zugriffsdistanzen, für welche das Multi-Workgroup-Tiling einen signifikanten Speedup gegenüber dem Single-Workgroup-Tiling und der jeweiligen Referenzimplementierung nur mit Kernelfusion, aber ohne Tiling erzielen kann. Zum Beispiel wird das Multi-Workgroup-Tiling bei dem Verner-Verfahren schneller als das Single-Workgroup-Tiling für Zugriffsdistanzen größer als 256. Gegenüber der Referenzimplementierung erzielt das Multi-Workgroup-Tiling bei dem Verner-Verfahren einen Speedup von 2.4 bei einer Zugriffsdistanz von 10 240, einen Speedup von 1.3 bei einer Zugriffsdistanz von 245 760, und einen Speedup kleiner als 1 erst für Zugriffsdistanzen, die größer als ungefähr 327 680 sind. Dieser große Bereich von Zugriffsdistanzen, für welche das Multi-Workgroup-Tiling einen deutlichen Speedup gegenüber der Referenzimplementierung erzielen kann, ist eine bemerkenswerte Verbesserung gegenüber dem Single-Workgroup-Tiling, welches gegenüber der Referenzimplementierung nur einen Speedup für Zugriffsdistanzen, die kleiner als 1 536 sind, erzielen kann. Leider zeigen die Experimente auch, dass das Multi-Workgroup-Tiling auf Kepler kaum lohnenswert ist, da es gegenüber der Referenzimplementierung nur einen kleinen Speedup von ungefähr 1.2 für Zugriffsdistanzen, die kleiner als 5 120 sind, erzielen kann. Der Speedup vom Multi-Workgroup-Tiling auf Polaris ist zwar größer als der Speedup vom Multi-Workgroup-Tiling auf Kepler, aber kleiner als der Speedup vom Multi-Workgroup-Tiling auf Volta. So erzielt das Multi-Workgroup-Tiling auf Polaris zum Beispiel einen Speedup von 2.6 für eine Zugriffsdistanz von 1 024 und noch einen Speedup von 1.5 für eine Zugriffsdistanz von 10 240. Jedoch hätte das Multi-Workgroup-Tiling sehr wahrscheinlich auch auf Polaris für mittelgroße und große Zugriffsdistanzen eine bessere Performance, wenn die im Abschnitt 13.3.2 dokumentierten Probleme mit Polaris unter OpenCL nicht aufgetreten wären.

Wie beeinflusst die Stufenanzahl des RK-Verfahrens die Performance des Multi-Workgroup-Tilings?
(Siehe Abbildung 10.14 und Abbildung 10.15)

Für kleine Zugriffsdistanzen ist der Speedup des Multi-Workgroup-Tilings umso größer, je mehr Stufen ein RK-Verfahren besitzt. Dies können wir dadurch erklären, dass RK-Verfahren

mit mehr Stufen auch eine niedrigere arithmetische Intensität besitzen. Zum Beispiel erzielt das Multi-Workgroup-Tiling auf Volta für eine Zugriffsdistanz von 32 einen Speedup von 4.8 für das Verner-Verfahren und einen Speedup von 3.6 für das Bogacki-Shampine-Verfahren. Zusätzlich würde man für große Zugriffsdistanzen erwarten, dass das Multi-Workgroup-Tiling weniger effizient wird, umso mehr Stufen ein Verfahren besitzt, da die Anzahl an Stufen auch den Workingset vergrößert. Dennoch zeigen die Messungen, dass sowohl für das Verner-Verfahren mit acht Stufen als auch für das Bogacki-Shampine-Verfahren mit vier Stufen das Multi-Workgroup-Tiling für große Zugriffsdistanzen einen ähnlichen Speedup gegenüber der jeweiligen Referenzimplementierung erzielt.

Welches Tiling-Schema ist besser?
(Siehe Abbildung 10.14, Abbildung 10.15, Abbildung 10.16 und Abbildung 10.17)

Auf Volta erzielt sowohl beim Single-Workgroup-Tiling als auch beim Multi-Workgroup-Tiling das trapezoidale Tiling-Schema für kleine Zugriffsdistanzen eine bessere Performance als das hexagonale Tiling-Schema, während für größere Zugriffsdistanzen (zum Beispiel $d \geq 10\,240$ für das Verner-Verfahren oder $d \geq 1\,6384$ für das Bogacki-Shampine-Verfahren) das hexagonale Tiling-Schema eine bessere Performance erzielt. Auf Polaris ist beim Single-Workgroup-Tiling das trapezoidale Tiling-Schema besser für kleinere Zugriffsdistanzen und das hexagonale Tiling-Schema besser für größere Zugriffsdistanzen geeignet, während auf Polaris beim Multi-Workgroup-Tiling das hexagonale Tiling-Schema immer deutlich performanter als das trapezoidale Tiling-Schema ist.

Wie beeinflusst beim Multi-Workgroup-Tiling die Tile-Höhe in Abhängigkeit von der Zugriffsdistanz die Performance?
(Siehe Abbildung 10.20)

Für kleinere Zugriffsdistanzen sind größere Tile-Höhen effizienter, da sie das Datenvolumen stärker reduzieren. Jedoch verursachen größere Tile-Höhen auch einen größeren Workingset und skalieren folglich schlechter mit der Zugriffsdistanz. Im Gegensatz dazu ergeben kleinere Tile-Höhen kleinere Speedups für kleinere Zugriffsdistanzen, aber sie skalieren auch besser mit der Zugriffsdistanz.

Wie beeinflusst beim Multi-Workgroup-Tiling die Tile-Breite und die Anzahl an nebenläufigen Tiles die Performance in Abhängigkeit von der Zugriffsdistanz?
(Siehe Abbildung 10.18 und Abbildung 10.20)

Für alle Tile-Höhen gilt, dass die beste Anzahl an nebenläufigen Tiles mit der Zugriffsdistanz abnimmt, während die beste Tile-Breite mit der Zugriffsdistanz zunimmt. Dies können wir dadurch erklären, dass bei einer großen Anzahl von kleinen nebenläufigen Tiles der Overhead der Multi-Workgroup-Barrieren geringer und die Instruktionsmischung weniger synchronisiert ist. Jedoch kann eine große Anzahl an nebenläufigen, kleinen Tiles das Datenvolumen weniger stark reduzieren als eine geringe Anzahl an nebenläufigen, großen Tiles, während bei größeren Zugriffsdistanzen zusätzlich mehr Datenvolumen anfällt. Deshalb eignet sich eine große Anzahl an nebenläufigen, kleinen Tiles weniger gut für große Zugriffsdistanzen. Des Weiteren gilt, dass je größer die Tile-Höhe ist, umso größer ist auch die optimale Tile-Breite, und umso geringer ist die optimale Anzahl an nebenläufigen Tiles. Dies können wir dadurch erklären, dass für eine feste Tile-Breite bei größeren Tile-Höhen auch die äußeren RHS $\rightarrow$ LC-Glieder eines Tiles schmäler werden, wodurch das Tile nicht mehr alle seine Rechenkerne bei seinen äußeren RHS $\rightarrow$ LC-Gliedern ausnutzen kann. Dadurch werden Tiles mit geringen Breiten und großen Höhen weniger effizient. Die Messungen zeigen auch, dass in Abhängigkeit von der

gewählten Tile-Höhe es beim hexagonalen Tiling-Schema entweder besser ist, für die geraden Spalten aus Hexagonen eine große Tile-Breite und für die ungeraden Spalten eine geringe Tile-Breite zu wählen, oder vice versa. Einen ähnlichen Effekt können wir auch beim trapezoidalen Tiling-Schema beobachten, bei welchem es je nach der gewählten Tile-Höhe entweder besser ist, bei den geraden Trapezoiden eine große Tile-Breite und bei den ungeraden Trapezoiden eine geringe Tile-Breite zu wählen, oder vice versa. Des Weiteren verläuft die beste Anzahl an nebenläufigen Tiles in Abhängigkeit von der Zugriffsdistanz nicht glatt, sondern ihr Verlauf zeigt deutliche Spitzen. Dies ist darauf zurückzuführen, dass viele teils sehr unterschiedliche Tile-Breitenkonfigurationen für kleine bis mittelgroße Zugriffsdistanzen eine sehr ähnliche Performance liefern. Dennoch liefert eine bestimmte Tile-Breitenkonfiguration für ähnliche Zugriffsdistanzen immer auch eine sehr ähnliche Performance.

Wie viele Blöcke pro Thread und welche Strides liefern die beste Performance?
(Siehe Abbildung 10.19)

Für Zugriffsdistanzen kleiner als 32 768 erzielen ein Block per Thread und alle drei Strides mit zwei Blöcken pro Thread ungefähr die gleiche Performance. Für Zugriffsdistanzen größer als 32 768 sind zwei, drei oder vier Blöcke per Thread performanter als ein Block per Thread. Des Weiteren gilt für diejenigen Zugriffsdistanzen, für welche mehr als ein Block pro Thread performanter ist, dass Stride-TILE, welcher die Lastbalancierung auf Kosten der Lokalität bevorzugt, die beste Performance liefert.

Wie nutzen die Implementierungsvarianten die Ressourcen der GPU aus?
(Siehe Abbildung 10.21)

Das Profiling auf Volta für das Verner-Verfahren zeigt, dass die Referenzimplementierung stark durch die DRAM-Bandbreite gebunden ist. Deshalb wird der Speedup des Single- und Multi-Workgroup-Tilings für kleine Zugriffsdistanzen gegenüber der Referenzimplementierung dadurch erzielt, dass das Tiling das DRAM-Volumen stark reduzieren kann. Jedoch kann selbst das Single-Workgroup-Tiling, das eine schnelle Synchronisation über Single-Workgroup-Barrieren und einen schnellen Datentransfer für LC $\rightarrow$ RHS-Abhängigkeiten über den Scratchpad-Speicher verwendet, bei kleinen Zugriffsdistanzen die DP-Rechenwerke nur zu 22 % und die IPC nur zu 24 % aber die DRAM-Bandbreite zu 89 % auslasten. Da beim Multi-Workgroup-Tiling die Synchronisation und der Datentransfer über den L2-Cache für die LC $\rightarrow$ RHS-Abhängigkeiten weniger effizient ist, kann es für kleine Zugriffsdistanzen die DP-Rechenwerke nur zu 20 % und die IPC zu nur 27 % und die DRAM-Bandbreite nur zu 59 % auslasten. Für kleine Zugriffsdistanzen vermindert dies beides den maximalen Speedup des Multi-Workgroup-Tilings im Vergleich zu Single-Workgroup-Tiling. Für große Zugriffsdistanzen kann das Multi-Workgroup-Tiling zudem das DRAM-Volumen nicht mehr effizient reduzieren, wodurch sich die Auslastung der DRAM-Bandbreite erhöht. Dadurch wird das Multi-Workgroup-Tiling sehr stark durch die DRAM-Bandbreite gebunden, und lastet die DP-Rechenwerke der GPU nur noch zu 5 % aus.

Wieso lässt sich mit Multi-Workgroup-Tiling auf Kepler ein deutlich schlechterer Speedup als auf Volta erzielen, und wieso liegt auf Polaris der durch das Multi-Workgroup-Tilings erzielte Speedup dazwischen?
(Siehe Abbildung 10.21 und Abbildung 10.22)

Volta besitzt einen 20 480 KiB großen Registersatz sowie einen 6 144 KiB großen L2-Cache, Polaris einen 9 216 kiB großen Registersatz sowie einen 1 024 KiB großen L2-Cache und Kepler einen 3 840 KiB großen Registersatz sowie einen 1 536 KiB großen L2-Cache. Diese klei-

neren On-Chip-Speicher limitieren die Datenwiederverwendung auf Kepler und Polaris, und vermindern dadurch die Effizienz des Multi-Workgroup-Tilings. Deshalb reduziert auf Kepler das Multi-Workgroup-Tiling das DRAM-Volumen der Referenzimplementierung nur von 14 auf 7 GB, während das Multi-Workgroup-Tiling das DRAM-Volumen auf Volta von 14 GB auf 2 GB reduziert. Des Weiteren ist die DRAM-Bandbreite auf Volta und Polaris ein engerer Flaschenhals als auf Kepler (Volta: 0.139 Bytes per DP-FLOP, Polaris: 0.043 Bytes per SP-FLOP, und Kepler: 0.196 Bytes per DP-FLOP), wodurch auch auf Volta und Polaris die Wiederverwendung von Daten wichtiger wird als auf Kepler. Dies wird durch die Ergebnisse des Profilings gestützt, die eine Ausnutzung der DP-Rechenwerke durch die Referenzimplementierung von ungefähr 5 % auf Volta und 9 % auf Kepler zeigen. Diese beiden Faktoren machen nicht nur das Single-Workgroup-Tiling, sondern auch das Multi-Workgroup-Tiling auf Volta viel lohnenswerter als auf Kepler. Da man davon ausgehen kann, dass weiterhin auf zukünftigen GPU-Generationen der Flaschenhals der Speicherbandbreite deutlich enger und die On-Chip-Speicher deutlich größer werden, wird Multi-Workgroup-Tiling auf zukünftigen GPU-Architekturen sehr wahrscheinlich noch einen größeren Speedup gegenüber einer Implementierungsvariante mit Kernelfusion erzielen können als auf aktuellen GPU-Architekturen.

Wie sehr lohnt sich das Tiling über die Zeitschritte verglichen mit dem Tiling nur über die Stufen?
(siehe Abbildung 10.23, Abbildung 10.24 und Abbildung 10.25)

Die Messungen zeigen, wie erwartet, dass je mehr Stufen ein RK-Verfahren besitzt, umso weniger lohnt sich das Tiling über die Zeitschritte verglichen zum Tiling nur über die Stufen. So lässt sich beispielhaft auf Kepler bei dem Heun-Verfahren mit zwei Stufen durch das Tiling über die Zeitschritte bei einer kleinen Zugriffsdistanz von 32 ein Speedup von 2.89 gegenüber der Referenzimplementierung erzielen, während sich durch das Tiling nur über die Stufen nur ein Speedup von 1.41 erzielen lässt. Dahingegen lässt sich beispielhaft bei dem RK-3/8-Verfahren mit 4 Stufen sowohl durch das Tiling über die Stufen als auch durch Tiling über die Zeitschritte bei derselben Zugriffsdistanz von 32 ein Speedup von circa 2.85 gegenüber der Referenzimplementierung erzielen. Interessanterweise lässt sich bei dem Euler-Verfahren, obwohl es nur eine Stufe besitzt, durch das Tiling über die Zeitschritte bei derselben Zugriffsdistanz von 32 nur ein Speedup von 1.75 gegenüber der Referenzimplementierung erzielen. Dies ist vermutlich darauf zurückzuführen, dass die Referenzimplementierung vom Euler-Verfahren bereits eine deutlicher höhere arithmetische Intensität als die Referenzimplementierung vom Heun-Verfahren oder vom RK-3/8-Verfahren besitzt, wodurch der durch das Tiling erzielte Speedup auch geringer ausfällt.

10.4 Tiling auf CPUs

10.4.1 Implementierung

10.4.1.1 Allgemeines

Um das Tiling auf CPUs zu realisieren, implementierten wir das Single-Thread-Tiling (siehe Abbildung 10.26), das Multi-Threaded-Single-Core-Tiling und das Multi-Core-Tiling. Da sich das Multi-Threaded-Single-Core-Tiling und das Multi-Core-Tiling in ihrer Implementierung nur dadurch unterscheiden, wie das Kernel die Threads eines Tiles den logischen CPU-Kernen zuweist, bezeichnen wir sie im Folgenden allgemein als *Multi-Thread-Tiling* (siehe Abbildung 10.27). Für sämtliche dieser Tilings übersetzen wir wieder jedes abstrakte Kernel aus unseren theoretischen Überlegungen in ein paralleles Schleifennest, wobei die parallele äußerste Schleife über die Tiles iteriert.

Da wir die Kernel mit Tiling nicht allgemein für eine beliebige CPU-Architektur optimieren können, überlegen wir uns im Folgenden wieder nur, wie wir die Kernel mit Tiling für moderne x64-Server-CPUs, wie sie zum Beispiel im Abschnitt 3.14 beschrieben sind, optimieren können. Dabei berücksichtigen wir insbesondere die Vektorisierung und wie sich mit Non-Temporal-Store-Instruktionen der Fetch-On-Write-Overhead des Last-Level-Caches vermeiden lässt. Allerdings lassen sich analoge Überlegungen auch verwenden, um die Kernel mit Tiling für andere CPU-Architekturen zu optimieren.

Wegen der Komplexität des Tilings ist auf CPUs die Datenwiederverwendung innerhalb eines Tiles und der Datentransfer zwischen den Tiles zwar ähnlich, nichtsdestotrotz aber komplexer als innerhalb beziehungsweise zwischen den Blöcken bei den Varianten mit Fusion. So kann ein Tile die Caches der CPU verwenden, um das Ergebnis von einer fusionierten Vektorgrundoperation zu einer davon abhängigen fusionierten Vektorgrundoperation zu transferieren, oder um einen permanenten Argumentvektor aus dem DRAM wiederzuverwenden, der von mehreren fusionierten Vektorgrundoperationen des Tiles gelesen wird. Für den Datentransfer innerhalb eines Tiles verwenden wir wieder die Technik der *Array-Contraction*, so dass ein Tile die von ihm produzierten Vektorkomponenten mit Hilfe eines *temporären Arrays* zu den abhängigen Vektorgrundoperationen transferiert, wodurch das Tile diese produzierten Vektorkomponenten nicht unnötig in den DRAM zurückschreibt. Ebenso schreibt ein Tile die von ihm produzierten Vektorkomponenten per Non-Temporal-Store-Instruktionen in den permanenten Vektor zurück, um so den Fetch-On-Write-Overhead des Last-Level-Caches der CPU zu vermeiden. Wenn wir für das Tiling annehmen, dass die Caches der CPU hinreichend groß sind, so findet auf CPUs wieder nur der Datentransfer zwischen unterschiedlichen Tiles beziehungsweise Kernels über den DRAM statt, während der Datentransfer innerhalb eines Tiles komplett über die Caches der CPU stattfindet.

Um das Tiling auf CPUs zu implementieren, stellen sich uns folgende Hauptherausforderungen: Wir müssen Schleifen definieren, die den Datentransfer zwischen den Tiles handhaben, das heißt das Lesen von Komponenten aus dem DRAM und das Schreiben von Komponenten in den DRAM, und Schleifen, die die fusionierten Vektorgrundoperationen für das entsprechende Argumentintervall des Tiles berechnen. Zusätzlich müssen wir die einzelnen Schleifen so wählen beziehungsweise die Schleifen so kombinieren, dass das resultierende Schleifennest des Tiles eine möglichst hohe Speicherzugriffslokalität besitzt.

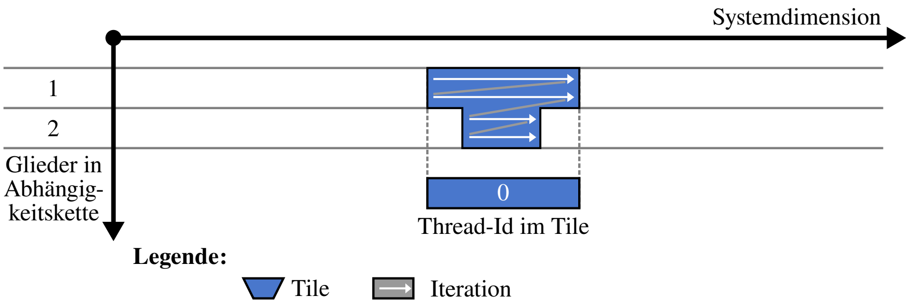


Abbildung 10.26: Abarbeitung eines Single-Thread-Tiles.

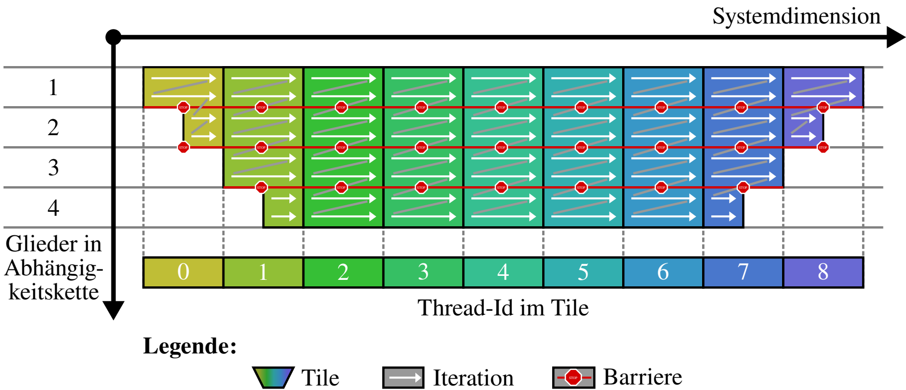


Abbildung 10.27: Abarbeitung eines Multi-Thread-Tiles.

10.4.1.2 Datentransfer zwischen Tiles und innerhalb eines Tiles

Auch für CPUs als Zielplattform müssen wir wieder bei denjenigen Komponenten einer Vektorgrundoperation, die zwischen den Tiles transferiert werden, die folgenden drei Fälle betrachten:

- keine Komponenten in einem Tile müssen zu einem anderen Tile transferiert werden,
- nur die Komponenten am rechten und linken Rand eines Tile müssen zu einem anderen Tile transferiert werden, oder
- alle Komponenten in einem Tile müssen zu einem anderen Tile transferiert werden.

Um dieses Transferieren zu verwirklichen, müssen wir nun diese drei Fälle in Schleifen und in Anweisungen umsetzen.

Als Erstes werden wir diejenigen Fälle, welche bestimmen, ob ein Tile die zur Auswertung einer Vektorgrundoperation benötigten Komponenten aus einem temporären Array oder einem permanenten Vektor laden soll, und deren Umsetzung behandeln:

- **Alle zu lesenden Vektorkomponenten wurden von dem Tile selbst produziert:** In diesem Fall wurden vom Tile vorher sämtliche zu lesenden Vektorkomponenten in einem temporären Array abgespeichert. Somit kann das Tile dieses Array der lesenden Vektorgrundoperation als Argument übergeben.
- **Keine der zu lesenden Vektorkomponenten wurden von dem Tile selbst produziert:** In diesem Fall sind alle der zu lesenden Vektorkomponenten in einem permanenten Vektor abgespeichert. Somit kann das Tile diesen permanenten Vektor der lesenden Vektorgrundoperation als Argument übergeben.
- **Nur die Vektorkomponenten in der Mitte des Tiles wurden von dem Tile selbst produziert:** In diesem Fall sind die mittleren zu lesenden Vektorkomponenten in einem temporären Array abgespeichert, während sich die zu lesenden Vektorkomponenten am linken und am rechten Rand des Tiles in einem permanenten Vektor im DRAM befinden. Für diesen Fall können wir uns drei mögliche Implementierungen überlegen:
  - **Explizite Kopierschleifen (siehe Abbildung 10.28):** Das Tile führt zuerst zwei *explizite Kopierschleifen* vor der Vektorgrundoperation aus, welche die linken und die rechten Komponenten von dem permanenten Vektor in das temporäre Array kopieren. Danach übergibt das Tile dieses temporäre Array der Vektorgrundoperation als Argument.
  - **Loop-Splitting (siehe Abbildung 10.29):** Die Schleife einer Vektorgrundoperation wird per *Loop-Splitting* in mehrere Schleifen aufgespalten, so dass jede Schleife ohne weitere Bedingungsanweisungen entweder aus einem temporären Array oder einem permanenten Vektor liest. Bei einer RHS-Operation benötigt dieses Loop-Splitting eine speziell angepasste Implementierung der Eval-Range-Funktion. Bei einer LC-Operation kann es vorkommen, dass das Tile jeweils unterschiedliche Bereiche von den Argumentvektoren dieser LC-Operation entweder aus einem permanenten Vektor oder einem temporären Array lesen muss. Dadurch müssen wir die Schleife der LC-Operation gegebenenfalls zusätzlich mehrmals aufspalten.
  - **Indexbasierte bedingte Anweisungen (siehe Abbildung 10.30):** Das Tile führt eine *indexbasierte bedingte Anweisung* für jede zu lesende Vektorkomponente aus, die die Vektorkomponenten in Abhängigkeit von ihrem Index entweder von dem permanenten Vektor oder dem temporären Array lädt. Dadurch benötigt dieser Ansatz für RHS-Operationen wieder eine spezielle Implementierung der Eval-Range- und Eval-Comp-Funktion.

  Da sowohl explizite Kopierschleifen als auch indexbasierte Anweisungen einen sehr großen Overhead besitzen, implementierten wir nur Loop-Splitting.

Als Nächstes werden wir diejenigen Fälle, welche bestimmen, ob ein Tile die produzierten Vektorkomponenten einer von ihm ausgewerteten Vektorgrundoperation in einem temporären Array und beziehungsweise oder in einem permanenten Vektor abspeichern soll, und deren Umsetzung behandeln:

- **Das Tile muss keine der produzierten Komponenten in den permanenten Vektor zurückschreiben und benötigt diese somit alle selbst zur Berechnung weiterer Vektorgrundoperationen:** In diesem Fall schreibt das Tile diese produzierten Komponenten über reguläre Store-Instruktionen direkt in ein temporäres Array.
- **Das Tile muss alle produzierten Komponenten in den permanenten Vektor zurückschreiben, aber benötigt keine dieser produzierten Komponenten zur Berechnung**

weiterer Vektorgrundoperationen: In diesem Fall schreibt das Tile die produzierten Komponenten direkt über Non-Temporal-Store-Instruktionen in den permanenten Vektor.

- **Das Tile muss alle der produzierten Komponenten in den permanenten Vektor zurückschreiben, benötigt diese aber zusätzlich selbst zur Berechnung weiterer Vektorgrundoperationen:** In diesem Fall schreibt das Tile diese produzierten Komponenten über reguläre Store-Instruktionen direkt in ein temporäres Array und über Non-Temporal-Store-Instruktionen direkt in den permanenten Vektor.
- **Das Tile muss die produzierten Komponenten an seinem rechten und linken Rand in den permanenten Vektor zurückschreiben, während es die produzierten Komponenten in seiner Mitte zur Berechnung weiterer Vektorgrundoperationen benötigt:** Für diesen Fall können wir uns drei mögliche Implementierungen überlegen:
  - **Explizite Kopierschleifen (siehe Abbildung 10.28):** Zuerst schreibt das Tile alle produzierten Vektorkomponenten über reguläre Store-Instruktionen in ein temporäres Array. Danach führt das Tile zwei zusätzliche *explizite Kopierschleifen* aus, welche die Komponenten am rechten Rand des Tiles und die Komponenten am linken Rand des Tiles von dem temporären Array in den permanenten Vektor über Non-Temporal-Store-Instruktionen kopieren.
  - **Loop-Splitting (siehe Abbildung 10.29):** Hier spalten wir per *Loop-Splitting* diejenige Schleife, die die Vektorgrundoperation berechnet, in drei Schleifen auf: Zuerst wertet die erste Schleife die linken Komponenten der Vektorgrundoperation aus, und schreibt die dabei produzierten Komponenten über reguläre Store-Instruktionen in ein temporäres Array und über Non-Temporal-Store-Instruktionen in den permanenten Vektor. Anschließend wertet die zweite Schleife die Vektorgrundoperation für die mittleren Komponenten aus und schreibt dabei die produzierten Komponenten über reguläre Store-Instruktionen nur in das temporäre Array. Anschließend wertet die dritte Schleife die rechten Komponenten der Vektorgrundoperation aus, und schreibt die dabei produzierten Komponenten über reguläre Store-Instruktionen in ein temporäres Array und über Non-Temporal-Store-Instruktionen in den permanenten Vektor. Hierbei ist es wichtig anzumerken, dass dieses Loop-Splitting beim Zurückschreiben mit dem Loop-Splitting beim Lesen interagiert, wodurch wir die Schleifen einer Vektorgrundoperation gegebenenfalls mehrmals aufspalten müssen.
  - **Index-basierte bedingte Anweisungen (siehe Abbildung 10.30):** Hier führt das Tile eine *Index-basierte bedingte Anweisung* für jede produzierte Komponente der Vektorgrundoperation aus, die diese Komponente über eine Non-Temporal-Store-Instruktion in einen permanenten Vektor oder über eine reguläre Store-Instruktion in das temporäre Array schreibt.

  Da die expliziten Kopierschleifen und die indexbasierten Anweisungen einen sehr großen Overhead besitzen, implementierten wir wieder nur das Loop-Splitting.

10.4.1.3 Schleifenstrukturen für das Single-Thread Tiling

Die einfachste Möglichkeit, um das Tiling auf CPUs umzusetzen, ist das Single-Thread-Tiling, bei welchem jeder Thread an seinem eigenen Tile arbeitet. Ähnlich wie bei den Varianten mit Fusion existiert beim Single-Thread-Tiling eine parallele äußere Schleife, die über die Tiles

Lesen von F_1 via expliziten Kopierschleifen

```
in: double* y_k, F_1;
temp: double* F_1_temp;
out: double* Y_2;

code:
parallel for each tile t
   //... central region of F_1_temp is computed by t

   for each component i in left border of Y_2 in t
      F_1_Temp[i_rel(t,i)] = F_1[i];

   for each component i in right border of Y_2 in t
      F_1_Temp[i_rel(t,i)] = F_1[i];

   for each component i of Y_2 in t
      Y_2[i] <- y_k[i] + a_21 * F_1_Temp[i_rel(t,i)];
```

Zurückschreiben von Y_2 via expliziten Kopierschleifen

```
in: double* y_k, F_1;
temp: double* F_1_temp, Y_2_temp;
out: double* Y_2;

code:
parallel for each tile t
   //... F_1 is computed by t

   for each component i of Y_2 in t
      Y_2_temp[i_rel(t,i)] = y_k[i] + a_21 * F_1[i];

   for each component i in left border of Y_2 in t
      Y_2[i] <- Y_2_temp[i_rel(t,i)];

   for each component i in right border of Y_2 in t
      Y_2[i] <- Y_2_temp[i_rel(t,i)];
```

Abbildung 10.28: Lesen und Zurückschreiben aus beziehungsweise in permanente Vektoren via explizite Kopierschleifen. Hierbei symbolisiert der Operator <− Non-Temporal-Store-Instruktionen.

Lesen von F_1 via Loop-Splitting

```
in: double* y_k, F_1;
temp: double* F_1_temp;
out: double* Y_2;

code:
parallel for each tile t
   //... central region of F_1_temp is computed by t

   for each component i in left border of Y_2 in t
      Y_2[i] <- y_k[i] + a_21 * F_1[i];

   for each component i in central region of Y_2 in t
      Y_2[i] <- y_k[i] + a_21 * F_1_temp[i_rel(t,i)];

   for each component i in right border of Y_2 in t
      Y_2[i] <- y_k[i] + a_21 * F_1[i];
```

Zurückschreiben von Y_2 via Loop-Splitting

```
in: double* y_k, F_1;
temp: double* F_1_temp, Y_2_temp;
out: double* Y_2;

code:
parallel for each tile t
   //... F_1_temp is computed by t

   for each component i in left border of Y_2 in t
      double Y_2_Reg =  y_k[i] + a_21 * F_1_temp[i];
      Y_2_temp[i_rel(t,i)] = Y_2_Reg;
      Y_2[i] <- Y_2_Reg;

   for each component i in central region of Y_2 in t
      Y_2_temp[i_rel(t,i)] <- y_k[i] + a_21 * F_1_temp[i];

   for each component i in right border of Y_2 in t
      double Y_2_Reg =  y_k[i] + a_21 * F_1_temp[i];
      Y_2_temp[i_rel(t,i)] = Y_2_Reg;
      Y_2[i] <- Y_2_Reg;
```

Abbildung 10.29: Lesen und Zurückschreiben aus beziehungsweise in permanente Vektoren via Loop-Splitting. Hierbei symbolisiert der Operator <− Non-Temporal-Store-Instruktionen.

Lesen von F_1 via Index-basierten bedingten Anweisungen

```
in: double* y_k, F_1;
temp: double* F_1_temp;
out: double* Y_2;

code:
parallel for each tile t
   //... central region of F_1_temp is computed by t

   for each component i of Y_2 in t
      double F_1_reg;

      if i in left or right border of Y_2 in t
         F_1_reg = F_1[i];

      if i in central region of Y_2 in t
         F_1_reg = F_1_temp[i_rel(t,i)];

      Y_2[i] <- y_k[i] + a_21 * F_1_reg;
```

Zurückschreiben von Y_2 via Index-basierten bedingten Anweisungen

```
in: double* y_k, F_1;
temp: double* F_1_temp, Y_2_temp;
out: double* Y_2;

code:
parallel for each tile t
   //... F_1_temp is computed by t

   for each component i of Y_2 in t
      double Y_2_reg = y_k[i] + a_21 * F_1_temp[i];
      Y_2_temp[i_rel(t,i)] = Y_2_reg;

      if i in left or right border of Y_2 in t
         Y_2[i] <- Y_2_reg;
```

Abbildung 10.30: Lesen und Zurückschreiben aus beziehungsweise in permanente Vektoren via Index-basierten bedingten Anweisungen. Hierbei symbolisiert der Operator <− Non-Temporal-Store-Instruktionen.

iteriert. In unserer Implementierung wertet ein Thread beim Single-Thread-Tiling die Glieder der Abhängigkeitskette seines Tiles eins nach dem anderen aus, wobei der Thread für jedes Glied ein separates Schleifennest ausführt. Dadurch gibt es keine Schleifen über die Glieder des Tiles. Um im Folgenden die Schleifenstrukturen des Single-Thread-Tilings möglichst übersichtlich vorzustellen, fokussieren wir uns auf die Auswertung der Vektorgrundoperationen durch ein Tile, und vernachlässigen die eventuellen zusätzlichen Schleifen, welche sich aus dem Loop-Splitting für das Einlesen und Zurückschreiben von Komponenten aus den permanenten Vektoren ergeben. Ebenso vernachlässigen wir die Verwendung der Array-Contraction und der Non-Temporal-Store-Instruktionen. So implementierten wir die folgenden Schleifenstrukturen für das Single-Thread-Tiling, die noch einmal in Abbildung 10.31 und Abbildung 10.32 illustriert werden:

- **Simple-Tiled:** In dieser *Simple-Tiled* Schleifenstruktur besteht ein Tile aus einer For-Schleife für jede verschmolzene Vektorgrundoperation. Jede Schleife wertet die Komponenten ihrer Vektorgrundoperation, die zum Tile gehören, der Reihe nach beginnend von der ersten Komponente bis zur letzten Komponente aus.
- **Simple-Tiled with Loop-Reversal:** Die *Simple-Tiled with Loop-Reversal* Schleifenstruktur ist ähnlich zu der Simple-Tiled Schleifenstruktur, mit dem Unterschied, dass, wenn das Tile mindestens einen Vektor zwischen zwei aufeinanderfolgenden Schleifen wiederverwendet, sie die Iterationsrichtung der zweiten Schleife umkehrt, das heißt die zweite Schleife wertet die Komponenten der entsprechenden Vektorgrundoperation beginnend von der letzten Komponente bis zur ersten Komponente aus. Hierdurch soll erreicht werden, dass, wenn nach der ersten Schleife nur die zuletzt produzierten Komponenten eines Vektors in der niedrigsten Cache-Stufe eines CPU-Kerns sind, der CPU-Kern diese Komponenten dann sofort in der zweiten Schleife aus der niedrigsten Cache-Stufe wiederverwenden kann, bevor er sie in eine höhere Cache-Stufe verdrängt. Dadurch besitzen bei dieser Schleifenstruktur die Speicherzugriffe des Tiles eine erhöhte zeitliche Lokalität.
- **In-Link-Subblocked:** Bei der *In-Link-Subblocked* Schleifenstruktur gibt es in einem Tile für jedes verschmolzene Glied der Abhängigkeitskette ein Schleifennest. Dieses Schleifennest unterteilt sein Glied der Abhängigkeitskette in Subblöcke, über die es in einer äußeren For-Schleife von dem ersten Subblock bis zu dem letzten Subblock iteriert. Diese For-Schleife, welche über die Subblöcke eines Glieds iteriert, beinhaltet für jede Vektorgrundoperation dieses Glieds eine innere For-Schleife, die die Vektorgrundoperation für die Komponenten des aktuellen Subblocks berechnet.
- **In-Link-Subblocked with Loop-Reversal:** Die am stärksten optimierte *In-Link-Subblocked with Loop-Reversal* Schleifenstruktur ist ähnlich zu der In-link-Subblocked Schleifenstruktur, mit demjenigen Unterschied, dass das Schleifennest von jedem zweiten Glied über die Subblöcke in umgekehrter Reihenfolge iteriert, das heißt beginnend von dem letzten Subblock zu dem ersten Subblock, während die inneren For-Schleifen immer noch von der ersten bis zur letzten Komponente des aktuellen Subblocks iterieren. Auf diese Weise soll die zeitliche Lokalität der Speicherzugriffe und damit die Performance auf eine ähnliche Weise erhöht werden, wie bei der Simple-Tiled with Loop-Reversal Schleifenstruktur.

Simple-Tiled Schleifenstruktur

```
in: double* y_k;
out: double* F_1, Y_2, F_2, Y_3;

code:
parallel for each tile t

   first link l1 of t:

      for each component i in l1
         F_1[i] = RHS_Eval_Comp(y_k, i);
      for each component i in l1
         Y_2[i] = y_k[i] + a_21 * F_1[i];

   second link l2 of t:

      for each component i in l2
         F_2[i] = RHS_Eval_Comp(Y_2, i);
      for each component i in l2
         Y_3[i] = y_k[i] + a_31 * F_1[i] + a_32 * F_2[i];
```

Simple-Tiled with Loop-Reversal Schleifenstruktur

```
in: double* y_k;
out: double* F_1, Y_2, F_2, Y_3;

code:
parallel for each tile t

   first link l1 of t:

      for each component i in l1 forward
         F_1[i] = RHS_Eval_Comp(y_k, i);
      for each component i in l1 reversed
         Y_2[i] = y_k[i] + a_21 * F_1[i];

   second link l2 of t:

      for each component i in l1 forward
         F_2[i] = RHS_Eval_Comp(Y_2, i);
      for each component i in l1 reversed
         Y_3[i] = y_k[i] + a_31 * F_1[i] + a_32 * F_2[i];
```

Abbildung 10.31: Pseudocode der Simple-Tiled-Schleifenstrukturen des Single-Thread-Tilings auf CPUs.

In-Link-Subblocked Schleifenstruktur

```
in: double* y_k;
out: double* F_1, Y_2, F_2, Y_3;

code:
parallel for each tile t

   first link l1 of t:
      for each block b in l1
         for each component i in b
            F_1[i] = RHS_Eval_Comp(y_k, i);
         for each component i in b
            Y_2[i] = y_k[i] + a_21 * F_1[i];

   second link l2 of t:
      for each block b in l2
         for each component i in b
            F_2[i] = RHS_Eval_Comp(Y_2, i);
         for each component i in b
            Y_3[i] = y_k[i] + a_31 * F_1[i] + a_32 *F_2[i];
```

In-Link-Subblocked with Loop-Reversal Schleifenstruktur

```
in: double* y_k;
out: double* F_1, Y_2, F_2, Y_3;

code:
parallel for each tile t

   first link l1 of t:
      for each block b in l1 forward
         for each component i in b
            F_1[i] = RHS_Eval_Comp(y_k, i);
         for each component i in b
            Y_2[i] = y_k[i] + a_21 * F_1[i];

   second link l2 of t:
      for each block b in l1 reversed
         for each component i in b
            F_2[i] = RHS_Eval_Comp(Y_2, i);
         for each component i in b
            Y_3[i] = y_k[i] + a_31 * F_1[i] + a_32 * F_2[i];
```

Abbildung 10.32: Pseudocode der In-Link-Subblocked-Schleifenstrukturen des Single-Thread-Tilings auf CPUs.

Dabei gilt wieder, dass bei allen Schleifenstrukturen die innerste Schleife die Komponenten einer Vektorgrundoperation mit einem Stride von eins auswertet. Aus diesem Grund kann der Compiler diese innere Schleife bei allen Varianten leicht abrollen und vektorisieren.

10.4.1.4 Schleifenstrukturen für das Multi-Thread Tiling

Um eine Schleifenstruktur für das Multi-Thread-Tiling zu definieren, müssen wir uns zuerst überlegen, welches Mapping zwischen Vektorkomponenten und Threads in einem Multi-Thread-Tile diese Schleifenstruktur umsetzen soll. Für ein Mapping zwischen Vektorkomponenten und Threads existieren folgende drei Möglichkeiten:

- **All-Same-Mapping:** In dem *All-Same-Mapping* wertet ein Thread des Tiles alle Vektorgrundoperationen für dieselben Komponenten aus. Deshalb treten bei diesem Mapping nur zwischen unterschiedlichen RHS $\rightarrow$ LC-Gliedern Abhängigkeiten zwischen den Threads eines Tiles auf. Jedoch treten innerhalb eines einzigen RHS $\rightarrow$ LC-Glieds keine Abhängigkeiten zwischen den Threads eines Tiles auf. Um Race-Conditions zu vermeiden, benötigt dieses Mapping folglich nur eine Barriere über die Threads des Tiles zwischen jedem verschmolzenen RHS $\rightarrow$ LC-Glied. Zudem kann bei diesem Mapping jeder Thread viele Daten über seine schnellen privaten Caches wiederverwenden.
- **Link-Same-Mapping:** In dem *Link-Same-Mapping* berechnet ein Thread des Tiles dieselben Komponenten für alle Vektorgrundoperation innerhalb eines RHS $\rightarrow$ LC-Glieds, aber unterschiedliche Komponenten für unterschiedliche RHS $\rightarrow$ LC-Glieder. Aus diesem Grund braucht dieses Mapping ebenfalls nur eine Barriere zwischen den RHS $\rightarrow$ LC-Gliedern des Tiles. Jedoch können bei diesem Mapping die Threads weniger Daten über ihre privaten Caches wiederverwenden als beim All-Same-Mapping.
- **All-Different-Mapping:** In dem *All-Different-Mapping* berechnet ein Thread unterschiedliche Komponenten für unterschiedliche Vektorgrundoperationen, sogar wenn diese sich im selben RHS $\rightarrow$ LC-Glied befinden. Unglücklicherweise verursacht das All-Different-Mapping bei verschmolzenen RHS $\rightarrow$ LC-Abhängigkeiten zusätzlich Abhängigkeiten zwischen den Threads des Tiles. Dadurch benötigen diese RHS $\rightarrow$ LC-Abhängigkeiten eine zusätzliche Barriere. Zusätzlich reduziert das All-Different-Mapping auch die Datenwiederverwendung über die privaten Caches des Threads. Deshalb ist dieses Mapping mangels seiner Vorteile nur eine theoretische Überlegung.

Wegen dieser Vorteile implementierten wir nur Schleifenstrukturen für das All-Same-Mapping. Zusätzlich können wir das All-Same-Mapping weiter verfeinern:

- **Block-Wise-All-Same-Mapping:** In dem *Block-Wise-All-Same-Mapping* wertet jeder Thread alle Vektorgrundoperationen jeweils nur für den einen selben Block von aufeinanderfolgenden Komponenten aus. Bei diesem Mapping ist es jedoch nachteilig, dass die Threads am Rand des Tiles weniger Komponenten berechnen müssen als die Threads in der Mitte des Tiles. Dies resultiert in einem Lastungleichgewicht, welches die Performance verringern kann.
- **Block-Cyclic-All-Same-Mapping:** In dem *Block-Cyclic-All-Same-Mapping* gibt es eine zusätzliche Block-zyklische Verteilung von Vektorkomponenten auf die Threads des Tiles, so dass ein jeder Thread mehrere Blöcke von aufeinanderfolgenden Komponenten für

alle verschmolzenen Vektorgrundoperationen mit einem größeren Stride zwischen diesen Blöcken berechnet. Wegen dieses Strides besitzt das Block-Cyclic-All-Same-Mapping verglichen mit dem Block-Wise-All-Same-Mapping ein besseres Lastgleichgewicht, aber auch eine geringere räumliche Lokalität der Speicherzugriffe

Als Nächstes müssen wir uns überlegen, wie wir das soeben genannte Block-Wise-All-Same-Mapping und Block-Cyclic-All-Same-Mapping in Schleifenstrukturen für das Multi-Thread-Tiling übersetzen können. Für diese beiden Mappings implementierten wir jeweils mehrere Schleifenstrukturen. Bei diesen Schleifenstrukturen gibt es wieder eine parallele äußere Schleife, die über die Tiles iteriert, wobei hier nun jeweils eine Gruppe von Threads an jedem Tile zusammenarbeitet. Eine solche Gruppe von Threads wertet dabei innerhalb eines Tiles die RHS $\rightarrow$ LC-Glieder der Abhängigkeitskette eins nach dem anderen aus, wobei sie für jeweils jedes RHS $\rightarrow$ LC-Glied separate Schleifennester verwendet. Dadurch gibt es wieder keine Schleifen über die Glieder eines Tiles. Um im Folgenden die Schleifenstrukturen des Multi-Thread-Tilings wieder möglichst übersichtlich vorzustellen, fokussieren wir uns wieder auf die Auswertungen der Vektorgrundoperationen durch ein Tile, und vernachlässigen die eventuellen zusätzlichen Schleifen, welche sich aus dem Loop-Splitting für das Einlesen und Zurückschreiben von Komponenten aus permanenten Vektoren ergeben. Ebenso vernachlässigen wir die Verwendung der Array-Contraction und der Non-Temporal-Store-Instruktionen. So implementierten wir für das Block-Wise-All-Same-Mapping die folgenden Schleifenstrukturen (siehe Abbildung 10.33):

- **Block-Wise without Subblocking:** Bei dieser *Block-Wise without Subblocking* Schleifenstruktur besteht jedes RHS $\rightarrow$ LC-Glied des Tiles aus einer Schleife für jede Vektorgrundoperation, in welcher jeder Thread diese Vektorgrundoperation für die Komponenten seines zugewiesenen Blockes auswertet.
- **Block-Wise with Subblocking:** Diese *Block-Wise with Subblocking* Schleifenstruktur unterteilt in jedem RHS $\rightarrow$ LC-Glied den Block eines jeden Threads weiter in Subblöcke. Deshalb besteht hier ein jedes RHS $\rightarrow$ LC-Glied aus einer äußeren Schleife, in welcher jeder Thread über die Subblöcke seines Blocks iteriert. Diese äußere Schleife beinhaltet eine innere Schleife für jede Vektorgrundoperation in diesem Glied, in welcher jeder Thread diese Vektorgrundoperation für die Komponenten seines aktuellen Subblocks auswertet.

Wir implementierten die folgenden Schleifenstrukturen für das Block-Cyclic-All-Same-Mapping (siehe Abbildung 10.34):

- **Block-Cyclic with separate Loops:** Bei dieser *Block-Cyclic with separate Loops* Schleifenstruktur besteht jedes Glied des Tiles aus einem Schleifennest für jede Vektorgrundoperation. Bei diesem Schleifennest iteriert eine äußere Schleife über die Blöcke der Block-zyklischen Verteilung. Diese äußere Schleife beinhaltet eine innere Schleife, die die Vektorgrundoperation des Schleifennests für die Komponenten dieses Blockes auswertet.
- **Block-Cyclic with fused Loops:** Bei dieser *Block-Cyclic with fused Loops* Schleifenstruktur besteht jedes Glied im Tile aus einem einzigen Schleifennest. Dieses Schleifennest beinhaltet zunächst eine äußere Schleife, die über die Blöcke der Block-zyklischen

Verteilung des Mappings iteriert. Für jede Vektorgrundoperation des Glieds beinhaltet dieses Schleifennest eine innere Schleife, die über die Komponenten des Blocks iteriert und dabei die entsprechende Vektorgrundoperation auswertet.

Des Weiteren können wir alle Schleifenstrukturen für das Multi-Thread-Tiling noch mit Loop-Reversal kombinieren, das analog wie beim Single-Thread-Tiling die Iterationsrichtung jeder zweiten innersten Schleife über die Systemkomponenten oder jeder zweiten mittleren Schleife über die Subblöcke umkehrt. Zusätzlich gilt bei allen Schleifenstrukturen für das Multi-Thread-Tiling wieder, dass die innerste Schleife jeweils über die Systemdimension mit einem Stride von eins iteriert. Folglich kann der Compiler wieder diese innerste Schleife abrollen und vektorisieren.

10.4.1.5 Wahl des Mappings zwischen den Threads und Multi-Thread-Tiles

Beim Multi-Thread-Tiling auf CPUs müssen wir die Anzahl an Threads, die an einem Tile zusammenarbeiten, und folglich die Anzahl an Tiles, die die CPU gleichzeitig berechnet, passend wählen, um eine gute Performance zu erzielen. Denn je mehr Threads an einem Tile zusammenarbeiten, umso weniger Tiles führt die CPU gleichzeitig aus, wodurch wiederum mehr Cache-Speicherplatz für jedes Tile verfügbar wird. Dies bewirkt wiederum, dass größere Tile-Größen umso effizienter werden, je mehr Threads an einem Tile zusammenarbeiten. Jedoch erhöhen mehr Threads pro Tile auch den Overhead der Synchronisation und die Homogenität der Instruktionsmischung, die die Threads zu einem gegebenen Zeitpunkt ausführen. Dies beides reduziert die Ressourcenauslastung umso stärker, je mehr Threads an einem Tile zusammenarbeiten.

Um die Ressourcenauslastung zu maximieren, sollten zudem so viele Threads an jedem Tile zusammenarbeiten, dass möglichst wenige Threads untätig bleiben. Das heißt, für die Anzahl an Threads $n_{\text{threadstile}}$, die an einem Tile zusammenarbeiten, die Anzahl an Tiles n_{tiles}, die gleichzeitig von der CPU berechnet werden, und die Anzahl an Threads $n_{\text{threadscpu}}$, die die CPU gleichzeitig ausführen kann, sollte folgende Gleichung erfüllt sein:

$$\left\lfloor \frac{n_{\text{threads}}}{n_{\text{threads per tile}}} \right\rfloor = n_{\text{concurrent tiles}} \quad \text{und} \quad \left\lfloor \frac{n_{\text{threads}}}{n_{\text{concurrent tiles}}} \right\rfloor = n_{\text{threads per tile}}. \tag{10.19}$$

Zusätzlich sollte das Mapping zwischen den Threads und den Tiles beim Multi-Thread-Tiling Folgendes beachten:

- Falls das Mapping SMT ausnutzen soll, so haben wir die Wahl zwischen zwei Möglichkeiten:
  - Weist das Mapping die Threads eines Kerns dem gleichen Tile zu, so erhöht dies die Speicherzugriffslokalität eines Kerns.
  - Weist das Mapping die Threads eines Kerns jeweils einem unterschiedlichen Tile zu, so senkt dies für den Kern die Homogenität der Instruktionsmischung und erlaubt es dem Kern, die Latenzen der Barrieren besser zu überbrücken.
- Das Mapping sollte die Cache-Hierarchie ausnutzen, das heißt diejenigen Threads, die sich eine Ebene in der Cache-Hierarchie teilen, sollten miteinander an demselben Tile zusammenarbeiten.

Block-Wise without Subblocking Schleifenstruktur

```
in: double* y_k;
out: double* F_1, Y_2, F_2, Y_3;

code:
group parallel for each tile t
   first link l1 of t:
      for each component i in my block of l1
         F_1[i] = RHS_Eval_Comp(y_k, i);
      for each component i in my block of l1
         Y_2[i] = y_k[i] + a_21 * F_1[i];
   group_barrier(t);
   second link l1 of t:
      for each component i in my block of l2
         F_2[i] = RHS_Eval_Comp(F_1, i);
      for each component i in my block of l2
         Y_3[i] = y_k[i] + a_31 * F_1[i] + a_32 * F_2[i];
   group_barrier(t);
```

Block-Wise with Subblocking Schleifenstruktur

```
in: double* y_k;
out: double* F_1, Y_2, F_2, Y_3;

code:
group parallel for each tile t
   first link l1 of t:
      for each subblock s in my block of l1
         for each component i in s
            F_1[i] = RHS_Eval_Comp(y_k, i);
         for each component i in s
            Y_2[i] = y_k[i] + a_21 * F_1[i];
   group_barrier(t);
   second link l1 of t:
      for each subblock s in my block of l2
         for each component i in s
            F_2[i] = RHS_Eval_Comp(Y_2, i);
         for each component i in s
            Y_3[i] = y_k[i] + a_31 * F_1[i] + a_32 * F_2[i];
   group_barrier(t);
```

Abbildung 10.33: Pseudocode der Schleifenstrukturen mit Blocke-Wise-Mapping für das Multi-Thread-Tiling auf CPUs. In diesem Pseudocode beschreibt `my block` denjenigen Block an Komponenten, die einem Thread durch das Block-Wise-Mapping zugewiesen wurden.

Block-Cyclic with separate Loops Schleifenstruktur

```
in: double* y_k;
out: double* F_1, Y_2, F_2, Y_3;

code:
group parallel for each tile t
   first link l1 of t:
      for each block b in my blocks of l1
         for each component i in b
            F_1[i] = RHS_Eval_Comp(y_k, i);
      for each block b in my blocks of l1
         for each component i in b
            Y_2[i] = y_k[i] + a_21 * F_1[i];
   group_barrier(t);
   second link l2 of t:
      for each block b in my blocks of l2
         for each component i in b
            F_2[i] = RHS_Eval_Comp(Y_2, i);
      for each block b in my blocks of l2
         for each component i in b
            Y_3[i] = y_k[i] + a_31 * F_1[i] + a_32 * F_2[i];
   group_barrier(t);
```

Block-Cyclic with fused Loops Schleifenstruktur

```
in: double* y_k;
out: double* F_1, Y_2, F_2, Y_3;

code:
group parallel for each tile t
   first link l1 of t:
      for each block b in my blocks of l1
         for each component i in b
            F_1[i] = RHS_Eval_Comp(y_k, i);
         for each component i in b
            Y_2[i] = y_k[i] + a_21 * F_1[i];
   group_barrier(t);
   second link l2 of t:
      for each block b in my blocks of l2
         for each component i in b
            F_2[i] = RHS_Eval_Comp(Y_2, i);
         for each component i in b
            Y_3[i] = y_k[i] + a_31 * F_1[i] + a_32 * F_2[i];
   group_barrier(t);
```

Abbildung 10.34: Pseudocode der Schleifenstrukturen mit Blocke-Cyclic-Mapping für das Multi-Thread-Tiling auf CPUs. Dabei beschreibt `my blocks` diejenigen Blöcke an Komponenten, die einem Thread durch das Block-Cyclic-Mapping zugewiesen wurden.

- Falls die CPU aus mehreren NUMA-Domains besteht, dann sollten die Threads von unterschiedlichen NUMA-Domains durch das Mapping nicht an einem Tile zusammenarbeiten, beziehungsweise jede NUMA-Domain sollte durch das Mapping nur an ihren eigenen Tiles arbeiten.

10.4.1.6 Aliasing der temporären Arrays zur Reduktion des Workingsets

Um den Workingset eines Tiles zu reduzieren, verwenden wir die temporären Arrays bereits innerhalb eines Tiles wieder, indem wir ihnen jeweils mehrere Vektorgrundoperationen zuweisen. Wenn in einem Tile nach der Berechnung einer Vektorgrundoperation das vom Tile produzierte Ergebnis einer anderen Vektorgrundoperation, das in einem temporären Array abgespeichert ist, nicht mehr benötigt, so kann das Tile das temporäre Array wiederverwenden, um das vom Tile produzierte Ergebnis einer anderen Vektorgrundoperation abzuspeichern. Um den Workingset zu reduzieren und um ein unnötiges Zurückschreiben dieser temporären Arrays in den DRAM zu vermeiden, verwenden wir eine *Last-In-First-Out-Schlange* (*LIFO-Schlange*) für die Neuzuweisung der temporären Arrays.

10.4.1.7 Barrieren für das Multi-Thread-Tiling

Die Schleifenstrukturen für das Multi-Thread-Tiling benötigen am Ende jedes RHS $\rightarrow$ LC-Glieds eine Software-Barriere, die die Threads eines jeden Tiles miteinander synchronisiert. Es existieren zwei grundlegend unterschiedliche Arten von Software-Barrieren auf CPUs:

- **Signalbasierte Barrieren:** Bei *signalbasierten Barrieren* legen sich diejenigen Threads, welche an der Barriere warten, schlafen, um von dem Betriebssystem beziehungsweise von dem Signal eines anderen Threads aufgeweckt zu werden, wenn sie die Barriere verlassen dürfen.
- **Barrieren mit Busy-Waiting:** Bei *Barrieren mit Busy-Waiting* lesen die wartenden Threads in einer Schleife kontinuierlich den Wert eines atomaren Zählers oder einer Flag aus, welche einen bestimmten Wert annehmen müssen, damit die wartenden Threads die Schleife und damit die Barriere wieder verlassen können.

Die von uns durchgeführten Voruntersuchungen zeigten, dass signalbasierte Barrieren, wie eine selbstimplementierte Barriere, welche wir über Bedingungsvariablen der Pthread-Bibliothek realisierten, oder die designierte Barriere der Pthreads-Bibliothek, eine hohe Latenz besitzen. Deswegen implementierten wir eine einfache Barriere mit Busy-Waiting. Diese verwendet einen atomaren Zähler, welchen die ankommenden Threads inkrementieren und diesen dann per Busy-Waiting in einer Schleife so lange abfragen, bis alle Threads der Barriere diesen Zähler inkrementiert haben. Allerdings existieren für CPUs noch weitere, besser optimierte Implementierungsvarianten von Barrieren mit Busy-Waiting, wie sie zum Beispiel in den Arbeiten [K36] und [K37] vorgestellt und untersucht wurden.

10.4.1.8 Codegenerator für CPU-Kernels mit Tiling

Wir implementierten wieder einen Codegenerator für CPU-Kernel mit Tiling, welcher ausgehend von einem Datenflussgraphen und einem Tiling automatisch den Quelltext für die CPU-Kernel in der Programmiersprache C erzeugt. Nach der Generierung des Quelltexts durch

den Codegenerator kann unser Framework zu seiner Laufzeit diesen Quelltext mit einem C-Compiler zunächst zu einer dynamischen Bibliothek kompilieren, diese dann laden und anschließend das Kernel darüber auf den Kernen der CPU ausführen.

Dabei beherrscht der Codegenerator sämtliche Optimierungen, die wir in diesem Kapitel vorstellten, das heißt, er kann Code für das Single-Thread-Tiling und für das Multi-Thread-Tiling jeweils mit den vorgestellten Schleifenstrukturen erzeugen. Dabei beherrscht unser Codegenerator auch das Loop-Splitting, um die Sonderbehandlung für das Lesen und Zurückschreiben von Vektorkomponenten aus beziehungsweise in permanente Vektoren am rechten und linken Rand eines Tiles durchzuführen. Zusätzlich sind im generierten Quelltext die verwendeten Blockgrößen und Subblockgrößen der Schleifenstrukturen Compilezeitkonstanten, so dass bei der späteren Kompilierung der verwendete C-Compiler diese Schleifenstrukturen abrollen oder vektorisieren kann. Des Weiteren beherrscht der Codegenerator die Datenwiederverwendung über temporäre Arrays beziehungsweise per Array-Contraction, so dass ein Kernel keine Daten unnötig in den DRAM zurückschreibt. Diese temporären Arrays verwendet er bei der Generierung des Quelltextes in einer LIFO-Schlange wieder, um so den Workingset des Kernels zu verkleinern. Ebenso setzt er das Schreiben des Kernels in permanente Vektoren per Non-Temporal-Store-Instruktionen um, für welche er je nach dem unterstützten Befehlssatz AVX2-Intrinsics oder AVX-512-Intrinsics verwendet. Zusätzlich versieht er sämtliche Zeiger wieder mit dem Schlüsselwort `restrict`, damit der verwendete C-Compiler bei der späteren Kompilierung des Kernels ein Aliasing zwischen den Kernelargumenten ausschließen und dadurch die Speicherzugriffe des Kernels optimieren kann.

10.4.2 Experimente

10.4.2.1 Aufbau der Experimente

Als Nächstes werden wir die mit Tiling erzeugten Implementierungsvarianten experimentell auf CPUs untersuchen. Damit unsere experimentellen Untersuchungen des Tilings auf GPUs und auf CPUs möglichst vergleichbar sind, verwenden wir einen Aufbau für unsere Experimente, der sehr ähnlich zu dem experimentellen Aufbau zur Untersuchung des Tilings auf GPUs ist:

Betrachtete ODE-Verfahren: Als ODE-Verfahren verwenden wir dieselben beiden expliziten eingebetteten RK-Verfahren, wie wir sie bereits für die experimentelle Untersuchung des Tilings auf den GPUs verwendeten: das Bogacki–Shampine-2(3)-Verfahren mit vier Stufen und das Verner-5(6)-Verfahren mit acht Stufen. Die Koeffizienten dieser beiden RK-Verfahren werden im Abschnitt 13.7 gezeigt.

Testproblem und Problemgröße: Für sämtliche Messungen mit der beschränkten Zugriffsdistanz verwenden wir wieder BRUSS2D als zu lösendes Anfangswertproblem. Da es sich bei BRUSS2D um einen 5-Punkt-Stencil auf einem 2D-Gitter mit 2 Werten pro Gitterzelle handelt, die wir in der Row-Major-Anordnung als Array-of-Structures abspeichern, besitzt es bei einer Kantenlänge von xmax Gitterzellen entlang der x-Achse eine beschränkte Zugriffsdistanz d von $d = 2 \cdot \text{xmax}$. Um den Einfluss der beschränkten Zugriffsdistanz auf die Performance zu untersuchen, verwenden wir wieder ein analoges Vorgehen wie für die experimentelle Untersuchung des Tilings auf GPUs: Wir setzen die Anzahl an Gitterzellen entlang der x-Achse auf $\text{xmax} = d/2$ Gitterzellen und passen die Anzahl an Gitterzellen entlang der y-Achse so an, dass die Problemgröße n insgesamt ungefähr konstant bleibt. Für diese Problemgröße n wählten wir den gleichen Wert wie bei den Messreihen des Tilings mit doppelter Gleitkommagenauigkeit auf GPUs, das heißt $n = 32 \cdot 1024 \cdot 1024$ Komponenten. Gemäß der Problemstellung nutzen wir wieder nur die beschränkte Zugriffsdistanz von BRUSS2D aus. Somit spannt BRUSS2D über die Systemdimension und die Zeitdimension einen zweidimensionalen Iterationsraum auf, den wir über unser Tiling ausnutzen können. Jedoch nutzen wir nicht die zweidimensionale räumliche Struktur des Stencils aus, welche für BRUSS2D einen dreidimensionalen Iterationsraum ergeben würde (x-, y- und Zeitdimension).

Referenzimplementierung: Als Referenzimplementierung für die Speedup-Berechnung verwenden wir eine per Kernelfusion generierte Implementierungsvariante des entsprechenden ODE-Verfahrens. Diese Referenzimplementierung verschmilzt somit nur (RHS | LC | MAP) $\rightarrow$ (LC | MAP | RED)-Abhängigkeiten, aber keine LC $\rightarrow$ RHS-Abhängigkeiten. Dadurch ist die Referenzimplementierung identisch zu den Varianten vom Typ-FUS für RK-Verfahren aus dem Abschnitt 9.2.8.

Betrachtete Implementierungsvarianten: Für die Evaluierung des Single-Thread-Tilings verwenden wir als einfachste Schleifenstruktur die Simple-Tiled Schleifenstruktur (*Single-STD*) sowie als am stärksten optimierte Schleifenstruktur die In-Link-Subblocked with Loop-Reversal Schleifenstruktur mit einer Subblockgröße von 1 024 (*Single-IMP*). Für die Evaluierung des Multi-Thread-Tilings verwenden wir als einfachste Schleifenstruktur die Block-Wise without Subblocking Schleifenstruktur (*Multi-STD*) sowie als am stärksten optimierte Schleifenstruktur die Block-Cyclic with fused Loops Schleifenstruktur mit vier Blöcken per Thread und mit Loop-Reversal (*Multi-IMP*). Da unsere beiden Test-CPUs zweifaches SMT beherrschen, führen wir beim Single-Thread-Tiling sowohl Messreihen durch, für welche wir die SMT-Funktionalität ausnutzen, als auch Messreihen, für welche wir SMT-Funktionalität nicht ausnutzen. Für die Messreihen des Single-Thread-Tilings mit SMT starten wir jeweils zwei

Threads pro Kern, so dass jeder Kern jeweils zwei nebenläufige Tiles berechnet. Für die Messreihen des Single-Thread-Tilings ohne SMT starten wir dagegen nur einen Thread pro Kern, so dass jeder Kern jeweils nur ein nebenläufiges Tile berechnet. Für die Messreihen des Multi-Thread-Tilings nutzen wir stets das SMT der CPUs aus, indem wir zwei Threads pro Kern starten, und lassen die beiden Threads eines Kerns immer an demselben Tile zusammenarbeiten.

Bestimmung der optimalen Tile-Breiten, Tile-Höhen und der optimalen Anzahl an Threads pro Multi-Thread-Teile: Bedauerlicherweise integrierten wir die CPU-Unterstützung noch nicht in unseren im Abschnitt 10.2.11 beschriebenen Autotuning-Ansatz. Deshalb führten wir für die Experimente eine vollständige Suche über eine vorgegebene Menge an Tile-Breiten und Tile-Höhen durch, um so die optimale Tile-Breite und Tile-Höhe innerhalb dieser Menge zu ermitteln. Zusätzlich bestimmt diese vollständige Suche noch für das Multi-Thread-Tiling für jedes Kernel eine optimale Anzahl an Threads beziehungsweise Kernen pro Tile. Allerdings, damit die Dauer der Suche und damit die Dauer für die Durchführung unserer Experimente auf Grund der kombinatorischen Explosion nicht so groß wird, nutzt diese vollständige Suche nicht die Freiheitsgrade bei den Tile-Breiten aus, sondern erzwingt, dass sämtliche Tiles in einer Implementierungsvariante jeweils die gleiche Breite besitzen.

Verwendete CPUs: Wir führten unsere Experimente auf zwei CPUs mit unterschiedlichen Architekturen durch:

- **Skylake Core i9-7980XE mit 8 $\times$ 16 GiB PC4-17000 DDR4-Modulen (*Skylake*):** Laufzeitmessungen mit dem Single-Thread-Tiling und dem Multi-Thread-Tiling für das Verner-Verfahren und das Bogacki-Shampine-Verfahren, Untersuchungen zur optimalen Tile-Höhe, zur optimalen Tile-Breite und zur optimalen Anzahl an nebenläufigen Tiles sowie ein ausführliches Profiling.
- **Ryzen 5950x mit 4 $\times$ 16 GiB PC4-23466 DDR4-Modulen (*Ryzen*):** Laufzeitmessungen mit dem Single-Thread-Tiling und dem Multi-Thread-Tiling.

Durch diese DRAM-Bestückung ergibt sich eine Burst-DRAM-Bandbreite von 68 GB/s auf Skylake und von 46.9 GB/s auf Ryzen. Die weiteren technischen Daten beider CPUs sind im Abschnitt 13.6 zu finden.

Verwendete Compiler und Compileroptionen: Wir benutzten für die Kompilierung unserer Kernels sowohl auf Skylake als auch auf Ryzen wieder den *ICC-Compiler* von Intel [W17] mit der höchsten Optimierungsstufe (`-O3`). Damit dieser Intel-Compiler unsere Kernels für Skylake optimiert und dabei vor allem die SIMD-Instruktionen von AVX-512 in unseren Kernels generiert, verwendeten wir für Skylake wieder dieselben Compiler-Optionen wie für die Kompilierung der Basisimplementierung und der Varianten mit Kernelfusion (`-march=skylake-avx512` und `-qopt-zmm-usage=high`). Für Ryzen wiesen wir den Compiler genauso wie für die Kompilierung der Varianten mit Kernelfusion an, Code mit der AVX2-SIMD-Befehlssatzerweiterung zu generieren (`-march=core-avx2`).

10.4.2.2 Messungen

Mit diesem Aufbau führten wir folgende Messreihen durch:

- **Untersuchungen zur Laufzeit und zum Speedup für das Tiling über die Stufen (Abbildung 10.35, Abbildung 10.36 und Abbildung 10.37):** Der Einfluss der Zugriffsdistanz auf die inverse Laufzeit und den Speedup für die beiden Tiling-Schemata mit Single-Thread-Tiling sowie Multi-Thread-Tiling und mit den optimalen Tile-Breiten, der optimalen Anzahl an nebenläufigen Tiles und den optimalen Tile-Höhen für das Verner-Verfahren auf Skylake, für das Bogacki-Shampine-Verfahren auf Skylake und für das Verner-Verfahren auf Ryzen.
- **Untersuchungen zu den Tile-Breiten und den nebenläufigen Tiles (Abbildung 10.38):** Die performanteste Tile-Breite und die performanteste Anzahl an nebenläufigen Tiles in Abhängigkeit von der Zugriffsdistanz für das trapezoidale Multi-Thread-Tiling und das hexagonale Multi-Thread-Tiling mit mehreren fest gewählten Tile-Höhen für das Verner-Verfahren auf Skylake.
- **Untersuchungen zu den Tile-Höhen (Abbildung 10.39):** Der Einfluss von unterschiedlichen Tile-Höhen auf den Speedup für das Multi-Thread-Tiling mit der optimalen Tile-Breite und der optimalen Anzahl an nebenläufigen Tiles in Abhängigkeit von der Zugriffsdistanz für das Verner-Verfahren auf Skylake.
- **Profiling (Abbildung 10.40):** Profiling der *Ausnutzung der DP-Peak-Performance* ($U_{\text{DP-PEAK}}$), der *Auslastung der IPC* (U_{IPC}), der *DRAM-Bandbreite* (U_{DRAM}) und des *DRAM-Volumens* (V_{DRAM}) für das Verner-Verfahren mit Multi-Thread-Tiling und der Multi-IMP-Schleifenstruktur auf Skylake. Eine genaue Beschreibung dieser Metriken ist im Abschnitt 13.4 zu finden.

Zu den Messungen ist Folgendes anzumerken:

- Wir wählten wieder immer für die Achse der Zugriffsdistanz eine logarithmische Skalierung, damit wir die Performance des Multi-Thread-Tilings für einen großen Bereich an Zugriffsdistanzen sinnvoll visualisieren können. Diese logarithmische Skalierung erweckt jedoch auf den ersten Blick den visuellen Eindruck, dass das Single-Thread-Tiling für einen großen Bereich an Zugriffsdistanzen mit dem Multi-Thread-Tiling konkurrieren kann. Tatsächlich lohnt sich aber das Single-Thread-Tiling gegenüber dem Multi-Thread-Tiling nur für Zugriffsdistanzen, die geringer als 256 sind, während das Multi-Thread-Tiling noch bei einer Zugriffsdistanz von 262 144 deutlich schneller als die Referenzimplementierung ist.

- Aus diesem Grund verzichten wir in dieser Arbeit auf eine ausgiebige Untersuchung des Single-Thread-Tilings.

- Aus historischen Gründen verwendeten wir weder bei den Varianten mit Tiling noch bei der Referenz-Implementierung die Enabling-Transformationen aus dem Kapitel 9.

- Da wir die CPU-Unterstützung noch nicht in unseren komplexeren Autotuning-Ansatz, den wir im Abschnitt 10.2.11 beschrieben, integrierten, nutzt unser für die Experimente verwendeter einfacherer Autotuning-Ansatz nicht sämtliche Freiheitsgrade der Tiling-Schemata aus, und evaluiert zudem nur wenige vielversprechende Tile-Breiten. Ebenso setzten wir die internen Parameter der Kernels auf einen festen Wert. Hierunter fallen im Wesentlichen die Blockgrößen der Schleifenstrukturen, und ob beim Multi-Thread-Tiling die Threads eines CPU-Kerns am selben Tile zusammenarbeiten sollen. Durch einen effizienteren Autotuning-Ansatz könnten wir hier noch deutlich mehr Möglichkeiten eva-

luieren, ohne dass eine kombinatorische Explosion auftreten würde. Dadurch würde sich wahrscheinlich die Performance der Varianten mit Tiling noch einmal deutlich erhöhen.

- Da wir in unseren Experimenten bei dem Multi-Thread-Tiling jeweils die beiden Threads eines Kerns an einem Tile zusammenarbeiten lassen, umfasst das Multi-Thread-Tiling in unseren Experimenten auch denjenigen Sonderfall des Multi-Threaded-Single-Core-Tilings, bei welchem jeweils ein jeder Kern mit all seinen Threads an seinem eigenen Tile arbeitet.

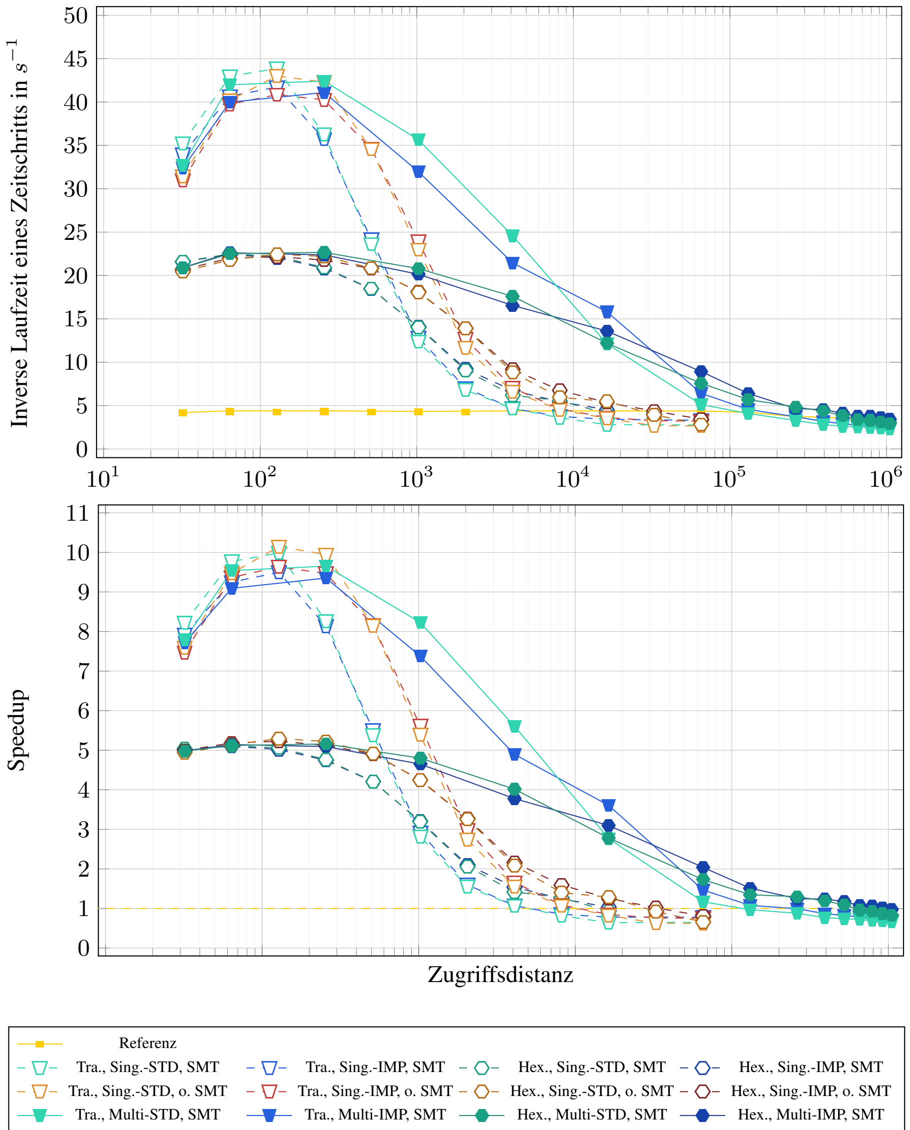


Abbildung 10.35: Laufzeituntersuchungen für das Verner-Verfahren mit Tiling auf Skylake. Diese Untersuchungen zeigen den Einfluss der Zugriffsdistanz auf die inverse Laufzeit und den Speedup für die beiden Tiling-Schemata mit den optimalen Tile-Breiten, den optimalen Tile-Höhen und der optimalen Anzahl an Threads pro Tile.

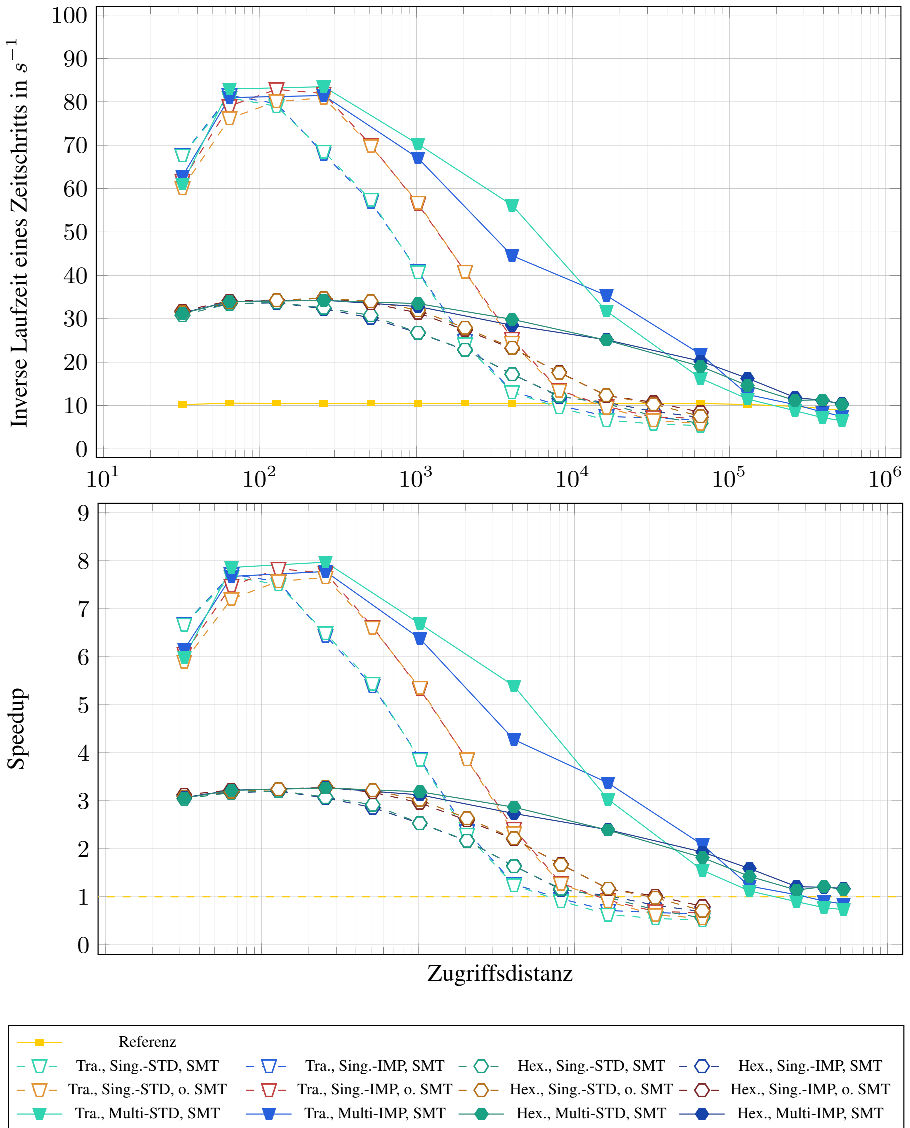


Abbildung 10.36: Laufzeituntersuchungen für das Bogacki-Shampine-Verfahren mit Tiling auf Skylake. Diese Untersuchungen zeigen den Einfluss der Zugriffsdistanz auf die inverse Laufzeit und den Speedup für die beiden Tiling-Schemata mit den optimalen Tile-Breiten, den optimalen Tile-Höhen und der optimalen Anzahl an Threads pro Tile.

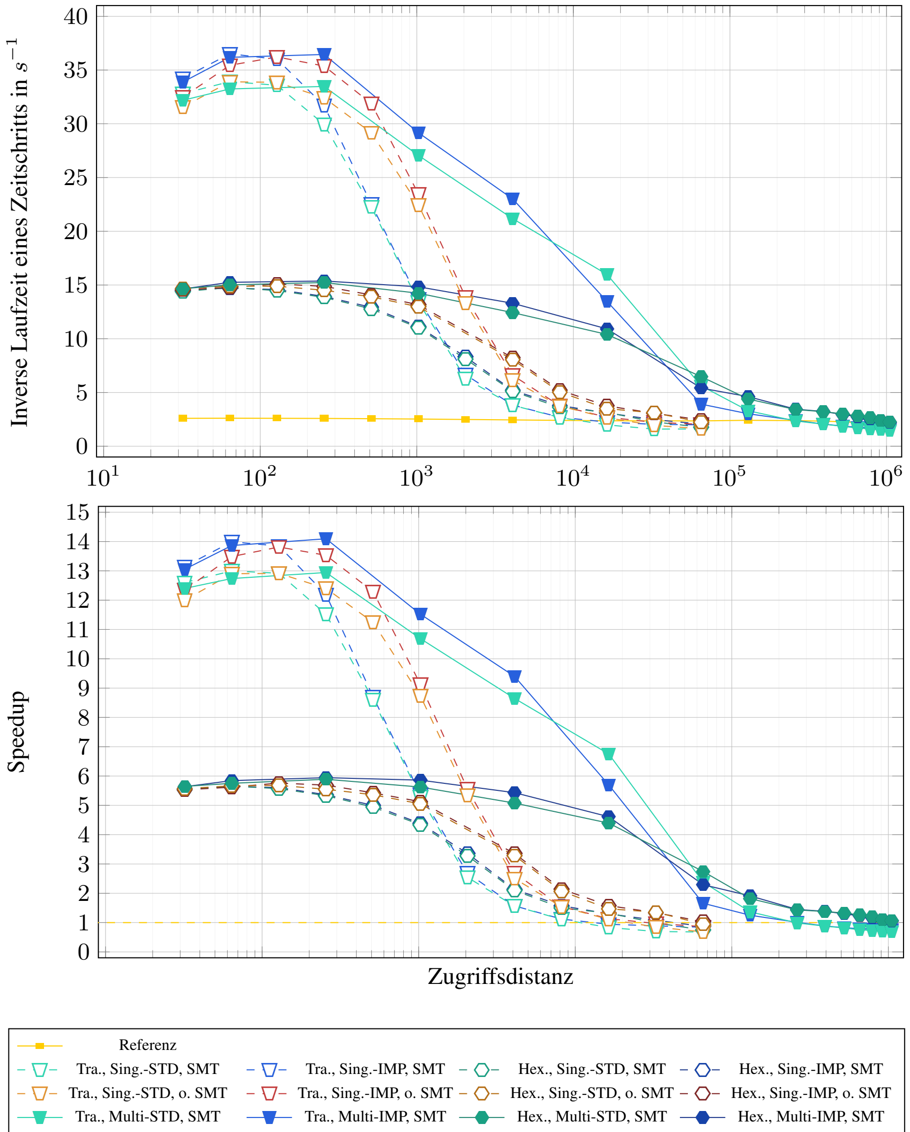


Abbildung 10.37: Laufzeituntersuchungen für das Verner-Verfahren mit Tiling auf Ryzen. Diese Untersuchungen zeigen den Einfluss der Zugriffsdistanz auf die inverse Laufzeit und den Speedup für die beiden Tiling-Schemata mit den optimalen Tile-Breiten, den optimalen Tile-Höhen und der optimalen Anzahl an Threads pro Tile.

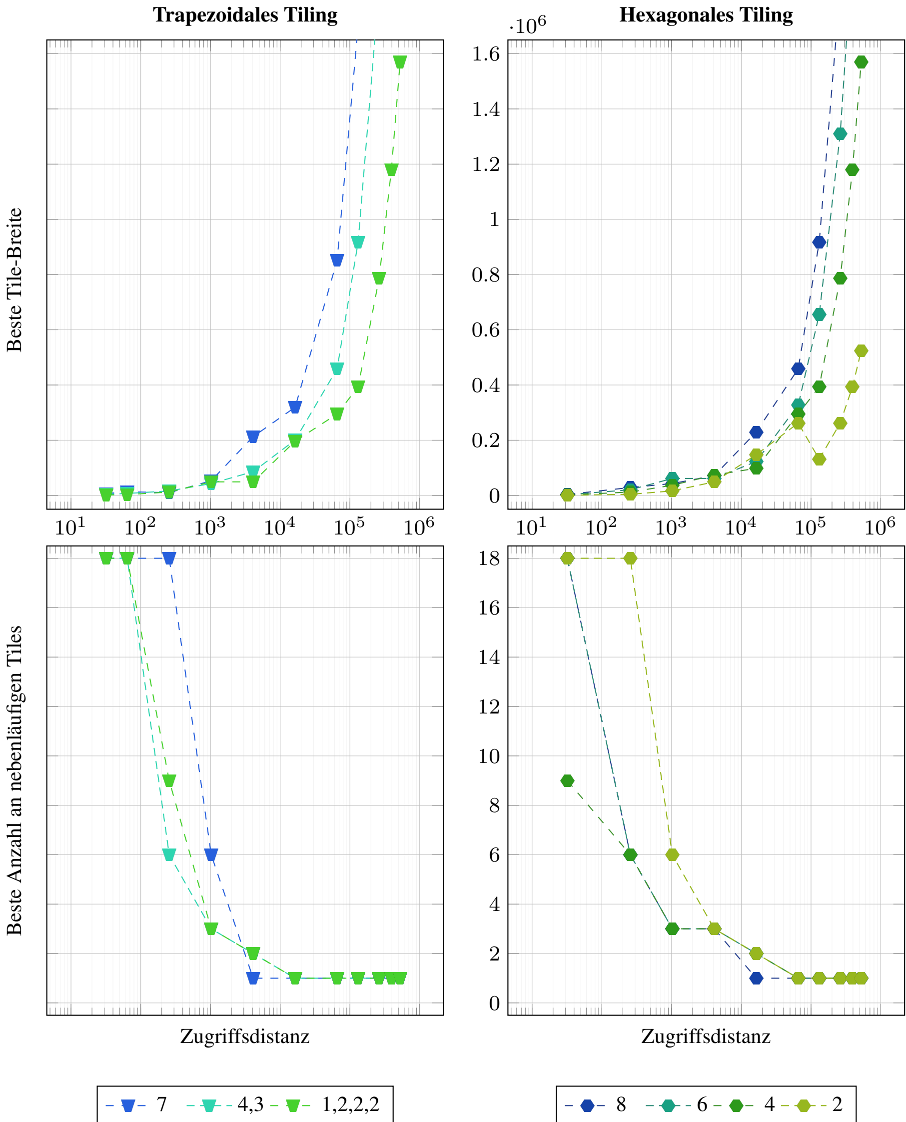


Abbildung 10.38: Untersuchungen zur besten Tile-Breite und der besten Anzahl an nebenläufigen Tiles beim Multi-Thread-Tiling. Hierbei untersuchen wir den Einfluss der Zugriffsdistanz auf die beste Tile-Breite und die beste Anzahl an nebenläufigen Tiles für mehrere fest gewählte Tile-Höhen (Legende) für das Verner-Verfahren und dem Multi-Thread-Tiling auf Skylake mit der IMP-Schleifenstruktur. Dabei geben die Einträge in der Legende für das trapezoidale Tiling die Tile-Höhen der Zeilen entlang der Abhängigkeitskette an.

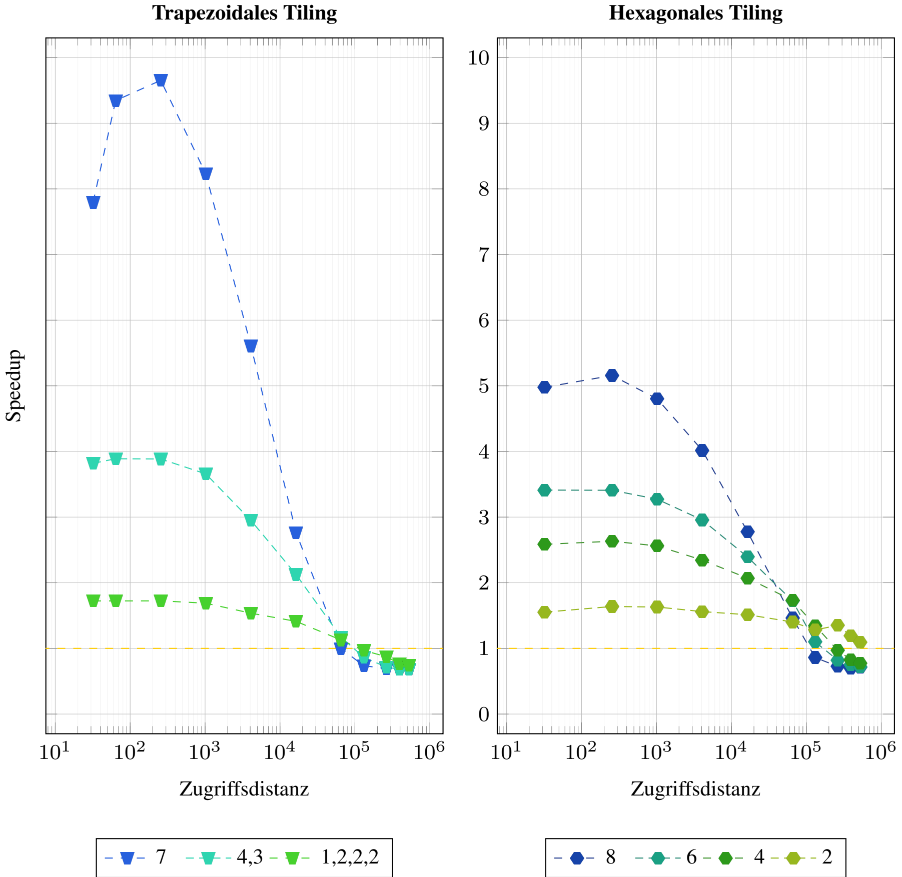


Abbildung 10.39: Untersuchungen zur besten Tile-Höhe beim Multi-Thread-Tiling. Diese Messung untersucht den Einfluss der Zugriffsdistanz auf den Speedup der beiden Tiling-Schemata für unterschiedliche Tile-Höhen mit der optimalen Tile-Breite mit dem Verner-Verfahren auf Skylake und der IMP-Schleifenstruktur. Dabei geben die Einträge in der Legende für das trapezoidale Tiling die individuellen Tile-Höhen entlang der Abhängigkeitskette an.

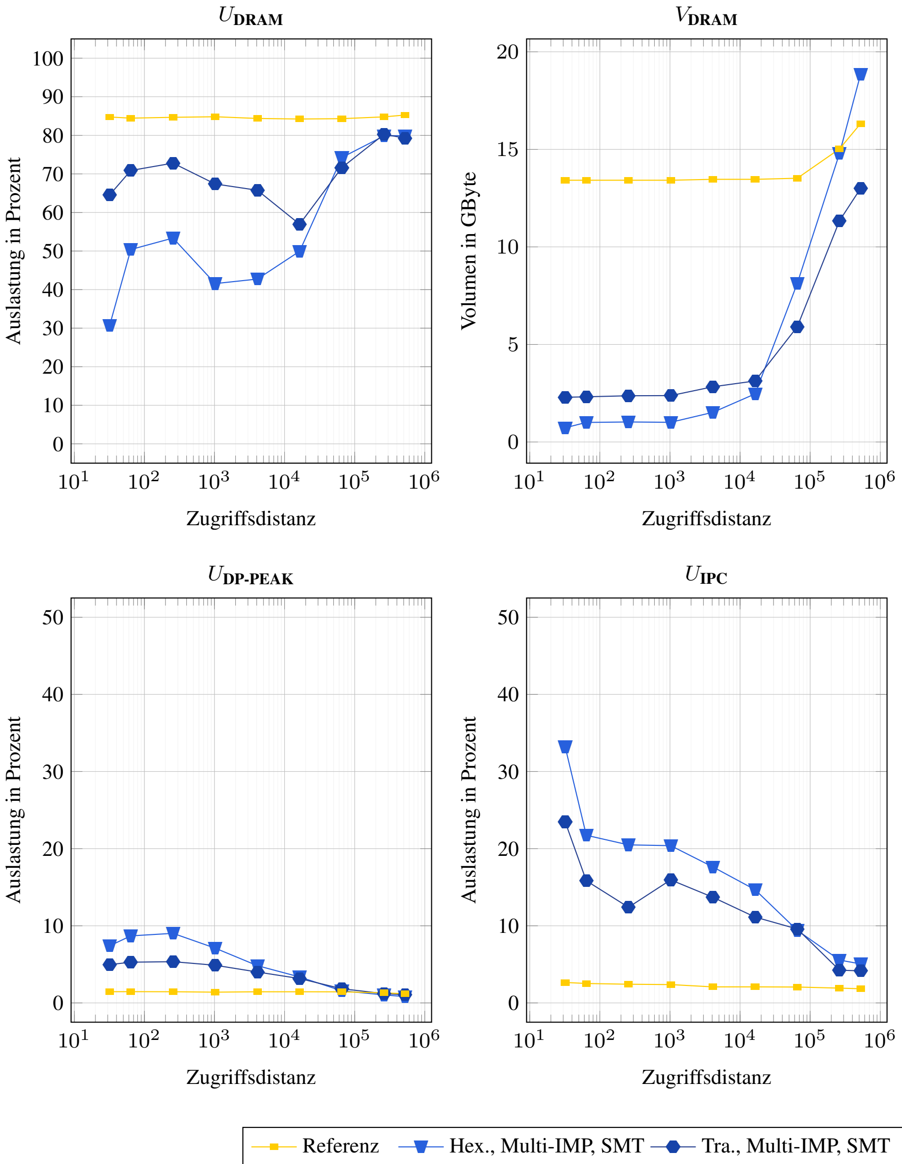


Abbildung 10.40: Profiling des Multi-Thread-Tilings auf Skylake. Dieses Profiling zeigt die Ausnutzung der DP-Peak-Performance ($U_{\text{DP-PEAK}}$) sowie die Auslastung der IPC (U_{IPC}), der DRAM-Bandbreite (U_{DRAM}) und das DRAM-Volumen (V_{DRAM}) für das Verner-Verfahren mit der IMP-Schleifenstruktur.

10.4.2.3 Diskussion

Mit Hilfe der Messungen können wir folgende Fragen beantworten:

Für welche Zugriffsdistanzen lohnt sich das Single-Thread-Tiling?
(Siehe Abbildung 10.35, Abbildung 10.36 und Abbildung 10.37)

Auf beiden betrachteten Test-CPUs und beiden betrachteten Verfahren lässt sich mit dem Single-Thread-Tiling bei sehr kleinen Zugriffsdistanzen ein großer Speedup gegenüber der Referenzimplementierung erzielen, wie zum Beispiel für das Verner-Verfahren bei einer beschränkten Zugriffsdistanz von 64 ein Speedup von 9.8 auf Skylake oder ein Speedup von 14.0 auf Ryzen. Allerdings ist das Single-Thread-Tiling selbst für kleine Zugriffsdistanzen nur in etwa genauso schnell wie das Multi-Thread-Tiling, welches zum Beispiel für das Verner-Verfahren bei einer Zugriffsdistanz von 64 einen Speedup von 9.5 auf Skylake oder einen Speedup von 13.8 Ryzen erzielt. Zudem skaliert das Single-Thread-Tiling deutlich schlechter mit der Größe der Zugriffsdistanz als das Multi-Thread-Tiling. So ist zum Beispiel das Single-Thread-Tiling für das Verner-Verfahren bei einer Zugriffsdistanz von bereits 32 768 nur noch so schnell wie die Referenzimplementierung, während das Multi-Thread-Tiling für diese Zugriffsdistanz noch einen Speedup von 2.6 erzielt. Somit lohnt sich das Single-Thread-Tiling gegenüber dem Multi-Thread-Tiling nur zum Lösen von Anfangswertproblemen mit einer sehr kleinen Zugriffsdistanz, wobei es auch bei solchen Anfangswertproblemen nur einen geringen Speedup gegenüber dem Multi-Thread-Tiling erzielen kann.

Für welche Zugriffsdistanzen lohnt sich das Multi-Thread-Tiling?
(Siehe Abbildung 10.35, Abbildung 10.36 und Abbildung 10.37)

Auf beiden betrachteten Test-CPUs und beiden betrachteten Verfahren lässt sich mit dem Multi-Thread-Tiling bei kleinen Zugriffsdistanzen ein großer Speedup gegenüber der Referenzimplementierung erzielen, wie zum Beispiel für das Verner-Verfahren bei einer beschränkten Zugriffsdistanz von 64 ein Speedup von 9.5 auf Skylake oder ein Speedup von 13.8 für auf Ryzen. Bereits bei diesen kleinen Zugriffsdistanzen ist das Multi-Thread-Tiling nur um circa 1 % bis 5 % langsamer als das Single-Thread-Tiling. Zudem skaliert das Multi-Thread-Tiling deutlich besser mit der Größe der Zugriffsdistanz als das Single-Thread-Tiling, wodurch es zum Beispiel für das Verner-Verfahren bei einer Zugriffsdistanz von 16 384 einen Speedup von 3.7 auf Skylake und von 6.8 auf Ryzen erzielt, während das Single-Thread-Tiling bei dieser Zugriffsdistanz nur noch einen Speedup von 2.2 auf Skylake und von 3.4 auf Ryzen erzielt. Des Weiteren lässt sich beim Verner-Verfahren mit Multi-Thread-Tiling bei einer sehr großen Zugriffsdistanz von circa 131 072 noch ein Speedup gegenüber der Referenzimplementierung von 1.6 auf Skylake und ein Speedup von 1.9 auf Ryzen erzielen, während es erst bei einer Zugriffsdistanz von circa 917 504 auf Skylake und von 1 048 576 auf Ryzen langsamer als die Referenzimplementierung wird. Durch diesen großen Bereich an Zugriffsdistanzen, in welchem das Multi-Thread-Tiling nicht nur einen signifikanten Speedup gegenüber dem Single-Thread-Tiling, sondern auch gegenüber der per Kernelfusion generierten Referenzimplementierung erzielt, stellt das Multi-Thread-Tiling eine deutliche Verbesserung gegenüber dem Single-Thread-Tiling dar.

Lohnt es sich bei dem Single-Thread-Tiling die SMT-Fähigkeit der Kerne von Skylake und Ryzen auszunutzen, indem wir nicht nur ein nebenläufiges Tile, sondern zwei nebenläufige Tiles auf jeden Kern berechnen lassen?
(Siehe Abbildung 10.35)

Die Untersuchungen zeigen, dass bei beiden betrachteten Verfahren und beiden betrachteten Systemen die Ausnutzung von SMT beim Single-Thread-Tiling die Performance für Zugriffs-

distanzen kleiner als 128 um bis zu 7 % erhöht. Dies ist wahrscheinlich darauf zurückzuführen, dass das Single-Thread-Tiling durch SMT bei diesen kleinen Zugriffsdistanzen seine Latenzen besser überbrücken kann. Jedoch skaliert das Single-Thread-Tiling mit SMT auch deutlich schlechter mit der Größe der Zugriffsdistanz als das Single-Thread-Tiling ohne SMT. So ist zum Beispiel beim Verner-Verfahren das Single-Thread-Tiling mit SMT bei einer Zugriffsdistanz von circa 2 048 nur noch in etwa so schnell wie die Referenzimplementierung, während das Single-Thread-Tiling ohne SMT bei dieser Zugriffsdistanz gegenüber der Referenzimplementierung noch einen Speedup von 1.4 erzielen kann. Dies können wir dadurch erklären, dass beim Single-Thread-Tiling mit SMT jedem Tile nur noch die Hälfte des On-Chip-Speichers zu Verfügung steht, und dass das Tiling bei größeren Zugriffsdistanzen auch größere Tiles benötigt, um das Datenvolumen effizient zu reduzieren. Diese Tiles besitzen jedoch auch einen größeren Workingset, welcher beim Single-Thread-Tiling mit SMT zu groß für die Caches der CPU wird.

Welchen Einfluss besitzen die Schleifenstrukturen auf die Performance des Tilings?
(Siehe Abbildung 10.35, Abbildung 10.36 und Abbildung 10.37)

Für das Single-Thread-Tiling gibt es in Abhängigkeit von der Zugriffsdistanz zwischen den einfacheren Single-STD-Schleifenstrukturen und den komplexeren Single-IMP-Schleifenstrukturen Laufzeitunterschiede von bis zu 10 %, während der Laufzeitunterschied beim Multi-Thread-Tiling zwischen den einfacheren Multi-STD-Schleifenstrukturen und den komplexeren Multi-IMP-Schleifenstrukturen sogar bis zu 25 % beträgt. Zudem gilt bei dem Multi-Thread-Tiling, dass oft die komplexere Multi-IMP-Schleifenstruktur deutlich langsamer als die einfachere Multi-STD-Schleifenstruktur ist. Jedoch gehen aus den Messungen keine einfachen Regeln hervor, welche Schleifenstruktur man für das Multi-Thread-Tiling und eine bestimmte Zugriffsdistanz wählen muss, um die maximale Performance zu erzielen, da die performanteste Schleifenstruktur zusätzlich von der verwendeten CPU und dem ODE-Verfahren abhängig ist.

Wie beeinflusst die Stufenanzahl des RK-Verfahrens die Performance des Multi-Thread-Tilings?
(Siehe Abbildung 10.35 und Abbildung 10.36)

Für kleine Zugriffsdistanzen ist der Speedup des Multi-Thread-Tilings umso größer, je mehr Stufen ein RK-Verfahren besitzt. Dies können wir dadurch erklären, dass RK-Verfahren mit mehr Stufen auch eine niedrigere arithmetische Intensität besitzen, wodurch sich wiederum durch die Reduktion des Datenvolumens über das Tiling ein größerer Speedup erzielen lässt. Zum Beispiel erzielt das Multi-Thread-Tiling auf Skylake für eine Zugriffsdistanz von 64 einen Speedup von 9.5 für das Verner-Verfahren und einen Speedup von 7.9 für das Bogacki-Shampine-Verfahren. Zusätzlich würde man für große Zugriffsdistanzen erwarten, dass das Multi-Thread-Tiling weniger effizient wird, umso mehr Stufen ein Verfahren besitzt, da die Anzahl an Stufen auch den Workingset vergrößert. Dennoch zeigen die Messungen, dass für beide untersuchte Verfahren das Multi-Thread-Tiling für große Zugriffsdistanzen einen ähnlichen Speedup gegenüber der Referenzimplementierung erzielt.

Welches Tiling-Schema ist besser?
(Siehe Abbildung 10.35, Abbildung 10.36 und Abbildung 10.37)

Die Untersuchungen zeigen, dass sowohl beim Single-Thread-Tiling als auch beim Multi-Thread-Tiling für kleinere Zugriffsdistanzen das trapezoidale Tiling-Schema eine geringere Laufzeit als das hexagonale Tiling-Schema besitzt. Jedoch skaliert das trapezoidale Tiling-

Schema auch schlechter mit der Größe der Zugriffsdistanz als das hexagonale Tiling-Schema, so dass für größere Zugriffsdistanzen das hexagonale Tiling-Schema eine geringere Laufzeit als das trapezoidale Tiling-Schema besitzt. Somit ist zum Beispiel beim Verner-Verfahren mit Multi-Thread-Tiling auf Skylake für Zugriffsdistanzen kleiner als 32 768 das trapezoidale Tiling-Schema performanter, während für Zugriffsdistanzen größer als 32 768 das hexagonale Tiling-Schema performanter ist.

Wie beeinflusst beim Multi-Thread-Tiling die Tile-Höhe in Abhängigkeit von der Zugriffsdistanz die Performance?
(Siehe Abbildung 10.39)

Für kleinere Zugriffsdistanzen sind größere Tile-Höhen effizienter, da sie das Datenvolumen stärker reduzieren können als kleinere Tile-Höhen. Jedoch verursachen größere Tile-Höhen auch einen größeren Workingset und skalieren folglich schlechter mit der Größe der Zugriffsdistanz. Im Gegensatz dazu ergeben kleinere Tile-Höhen kleinere Speedups für kleinere Zugriffsdistanzen, skalieren aber besser mit der Größe der Zugriffsdistanz.

Wie beeinflusst beim Multi-Thread-Tiling die Tile-Breite und die Anzahl an nebenläufigen Tiles die Performance in Abhängigkeit von der Zugriffsdistanz?
(Siehe Abbildung 10.38)

Für alle Tile-Höhen gilt, dass die beste Anzahl an nebenläufigen Tiles mit der Zugriffsdistanz abnimmt, während die beste Tile-Breite zunimmt. Dies können wir wieder dadurch erklären, dass bei einer großen Anzahl von kleinen nebenläufigen Tiles der Overhead der Barrieren eines Multi-Thread-Tiles geringer und die Instruktionsmischung weniger stark synchronisiert ist. Jedoch kann eine große Anzahl an nebenläufigen kleinen Tiles das Datenvolumen weniger stark reduzieren als eine geringe Anzahl an nebenläufigen großen Tiles, während bei größeren Zugriffsdistanzen auch mehr Datenvolumen anfällt. Deshalb eignet sich eine große Anzahl an nebenläufigen kleinen Tiles weniger gut für große Zugriffsdistanzen. Des Weiteren gilt, dass je größer die Tile-Höhe ist, umso größer ist auch die performanteste Tile-Breite und umso geringer ist die performanteste Anzahl an nebenläufigen Tiles. Dies können wir dadurch erklären, dass für eine feste Tile-Breite bei größeren Tile-Höhen auch die äußeren RHS $\rightarrow$ LC-Glieder des Tiles schmäler werden, wodurch das Tile auf Grund des von uns verwendeten All-Same-Mappings nicht mehr alle seine Rechenkerne bei seinen äußeren RHS $\rightarrow$ LC-Gliedern effizient ausnutzen kann. Dadurch werden Tiles mit geringen Breiten und großen Höhen weniger effizient.

Für welche Zugriffsdistanzen ist ein Multi-Core-Tiling besser geeignet als ein Single-Core-Tiling?
(Siehe Abbildung 10.35 und Abbildung 10.38)

Die Untersuchungen zeigen, dass beim Verner-Verfahren auf Skylake bis zu einer Zugriffsdistanz von 256 das Multi-Thread-Tiling mit 18 nebenläufigen Tiles am effizientesten ist. Bei 18 nebenläufigen Tiles führt das Multi-Thread-Tiling auf Skylake jedoch genau ein Tile mit zwei Threads pro Kern aus, wodurch es sich hier um ein Single-Core-Tiling handelt. Erst bei Zugriffsdistanzen größer als 256 wird das Multi-Thread-Tiling mit weniger als 18 nebenläufigen Tiles und mehr als einem Kern pro Tile, und damit das Multi-Core-Tiling, am performantesten.

Wie nutzen die Implementierungsvarianten die Ressourcen der CPU aus?
(Siehe Abbildung 10.40)

Das Profiling auf Skylake für das Verner-Verfahren zeigt, dass die Referenzimplementierung stark durch die DRAM-Bandbreite gebunden ist. Deshalb wird der Speedup des Multi-Thread-Tilings für kleine Zugriffsdistanzen gegenüber der Referenzimplementierung dadurch erzielt, dass das Multi-Thread-Tiling nur 5.2 % des DRAM-Volumens der Referenzimplementierung benötigt. Jedoch kann das Multi-Thread-Tiling selbst für kleine Zugriffsdistanzen die DP-Peak-Performance nur zu 9.1 % ausnutzen, und die IPC nur zu 33 % auslasten. Diese geringe Ausnutzung der Rechenressourcen ist für kleine Zugriffsdistanzen nicht mehr auf eine Limitierung durch die DRAM-Bandbreite zurückzuführen, da deren Auslastung bei kleinen Zugriffsdistanzen nur zwischen 30 % und 50 % beträgt. Deswegen führten wir weiteres Profiling durch, welches ergab, dass die Varianten mit Tiling nur circa 51 % aller Gleitkommaoperationen über AVX-512-Vektorinstruktionen durchführen, während sie die verbleibenden 49 % aller Gleitkommaoperationen über skalare Instruktionen durchführen. Eine vollständigere Vektorisierung würde die Performance somit für kleine Zugriffsdistanzen noch deutlich erhöhen. Für große Zugriffsdistanzen kann das Multi-Thread-Tiling das DRAM-Volumen gegenüber der Referenzimplementierung nicht mehr deutlich reduzieren, wodurch sich die Auslastung der DRAM-Bandbreite erhöht und wodurch die Ausnutzung der DP-Rechenleistung reduziert wird, so dass das Multi-Thread-Tiling ab einer Zugriffsdistanz von circa 131 072 durch die DRAM-Bandbreite gebunden wird.

Welche Unterschiede zeigen die experimentellen Untersuchungen zwischen Skylake und Ryzen auf?
(Siehe Abbildung 10.35 und Abbildung 10.37)

Bei einer kleinen Zugriffsdistanz von 64, bei welcher keine Limitierung durch die DRAM-Bandbreite besteht, zeigt sich, dass das Verner-Verfahren mit dem Single-Thread-Tiling oder dem Multi-Thread-Tiling auf Skylake einen Durchsatz von circa 43 Zeitschritten pro Sekunde und auf Ryzen einen Durchsatz von circa 37 Zeitschritten pro Sekunde erzielt. Somit ist das Verner-Verfahren mit Tiling bei kleinen Zugriffsdistanzen auf Skylake nur 1.16-mal so performant wie auf Ryzen. Da Skylake und Ryzen eine DP-Peak-Performance von 1.61 TFLOPS beziehungsweise 0.87 TFLOPS besitzen, deutet dies darauf hin, dass Skylake bei kleinen Zugriffsdistanzen, bei welchen keine Limitierung durch die DRAM-Bandbreite besteht, seine DP-Peak-Performance deutlich schlechter ausnutzen kann als Ryzen. Dies können wir über die unvollständige Vektorisierung der Varianten mit Tiling erklären und darüber, dass Skylake mit AVX-512 doppelt so breite Vektoreinheiten wie Ryzen mit AVX2 besitzt, und deshalb Skylake deutlich mehr von einer stärkeren Vektorisierung profitiert als Ryzen. Des Weiteren zeigen die Messungen, dass das Single-Thread-Tiling und das Multi-Thread-Tiling auf Skylake deutlich besser mit der Größe der Zugriffsdistanz als auf Ryzen skalieren. So besitzt zum Beispiel das Multi-Thread-Tiling beim Verner-Verfahren auf Skylake bei einer Zugriffsdistanz von 131 072 noch einen Durchsatz von 6.4 Zeitschritten pro Sekunde, während es auf Ryzen nur noch einen Durchsatz von 4.6 Zeitschritten pro Sekunde besitzt. Dies ist wahrscheinlich darauf zurückzuführen, dass Ryzen zwar insgesamt 1.58-mal so viel Cache-Speicherplatz wie Skylake besitzt (43.3 MiByte (L1 + L2 + L3) auf Skylake gegenüber 68.5 MiByte (L1 + L2 + L3) auf Ryzen), aber auch nur 69 % der DRAM-Bandbreite (68.0 GB/s auf Skylake gegenüber 46.9 GB/s auf Ryzen). Dadurch kann das Tiling auf Ryzen zwar gegenüber dem Tiling auf Skylake bei größeren Zugriffsdistanzen auch größere Tile-Größen verwenden, welche ein geringeres Datenvolumen besitzen und deren größeres Workingset noch in den Caches von Ryzen aber nicht mehr in

den Caches von Skylake Platz findet, jedoch kann vermutlich die geringere DRAM-Bandbreite von Ryzen das geringere Datenvolumen von den größeren Tiles nicht kompensieren. Auf Grund der größeren Caches und der geringeren DRAM-Bandbreite erzielt das Multi-Thread-Tiling allerdings bei großen Zugriffsdistanzen auf Ryzen einen größeren Speedup gegenüber der Referenzimplementierung als auf Skylake.

Welche Unterschiede und Gemeinsamkeiten zeigt die experimentelle Untersuchung des Tilings auf GPUs und CPUs?
(Siehe zusätzlich Abschnitt 10.3.2.3)

Zunächst werden wir das Single-Workgroup-Tiling auf GPUs mit dem Single-Thread-Tiling beziehungsweise mit dem Multi-Threaded-Single-Core-Tiling auf CPUs vergleichen, da diese beiden Tiling-Ansätze konzeptuell sehr ähnlich sind. Die Laufzeitmessungen zeigen, dass das Single-Workgroup-Tiling auf GPUs bei kleinen Zugriffsdistanzen einen etwas geringeren Speedup gegenüber der Referenzimplementierung erzielt als das Single-Thread-Tiling beziehungsweise das Multi-Threaded-Single-Core-Tiling auf CPUs (zum Beispiel 7.5 auf Volta gegenüber 9.5 Skylake bei einer Zugriffsdistanz von 64). Dies ist darauf zurückzuführen, dass die DRAM-Bandbreite auf den getesteten CPUs ein deutlich engerer Flaschenhals ist als auf den getesteten GPUs (0.0236 Bytes per DP-FLOP auf Skylake gegenüber auf 0.139 Bytes per DP-FLOP Volta), und somit durch die Reduktion des Datenvolumens über das Tiling potentiell ein größerer Speedup möglich ist. Die Laufzeitmessungen zeigen zusätzlich für kleine Zugriffsdistanzen, dass, trotz der unvollständigen Vektorisierung unserer Implementierung des Tilings auf CPUs, der Durchsatz an Zeitschritten sehr grob in Relation zur DP-Peak-Performance steht. So besitzt Volta in etwa die 3.8-fache DP-Peak-Performance wie Skylake und kann beim Verner-Verfahren mit dem Single-Workgroup-Tiling bei einer kleinen Zugriffsdistanz von 64 circa 7.7-mal so viele Zeitschritte pro Sekunde berechnen wie Skylake mit dem Single-Thread-Tiling. Diese nur sehr grobe Abhängigkeit lässt sich dadurch erklären, dass, wie wir durch das Profiling zeigten, Skylake die DP-Peak-Performance auf Grund der unvollständigen Vektorisierung deutlich schlechter ausnutzt als Volta.

Ebenso zeigen die Laufzeitmessungen, dass das Single-Workgroup-Tiling auf den untersuchten GPUs deutlich schlechter mit der Größe der Zugriffsdistanz skaliert als das Single-Thread-Tiling und das Multi-Threaded-Single-Core-Tiling auf den untersuchten CPUs. So ist das Single-Workgroup-Tiling auf Volta bereits bei einer Zugriffsdistanz von 2 048 so schnell wie die Referenzimplementierung, während das Single-Thread-Tiling auf Skylake bei dieser Zugriffsdistanz noch einen Speedup von 4.5 erzielen kann und erst bei einer Zugriffsdistanz von 32 768 so schnell wie die Referenzimplementierung wird. Dies können wir darauf zurückführen, dass auf den untersuchten GPUs jedem GPU-Kern deutlich weniger On-Chip-Speicher zur Verfügung steht als auf den untersuchten CPUs jedem CPU-Kern. So stehen zum Beispiel auf Volta jedem GPU-Kern nur 0.45 MiByte an On-Chip-Speicher (Registersatz + Scratchpad + L1 + L2) zur Verfügung, während auf Skylake jedem CPU-Kern 2.41 MiByte an On-Chip-Speicher (L1 + L2 + L3) zur Verfügung stehen. Aus diesem Grund ist diejenige Zugriffsdistanz, bei welcher auf GPUs das Multi-Workgroup-Tiling schneller als das Single-Workgroup-Tiling wird, beziehungsweise bei welcher auf CPUs das Multi-Core-Tiling schneller als ein Single-Core-Tiling wird, auf GPUs deutlich kleiner als auf CPUs. Diese Zugriffsdistanz beträgt zum Beispiel 192 auf Volta und 1 024 auf Skylake.

Zusätzlich zeigen die Laufzeitmessungen, dass das Multi-Core-Tiling auf GPUs und das Multi-Workgroup-Tiling auf CPUs sehr ähnlich mit der Größe der Zugriffsdistanz skalieren. So erzielen zum Beispiel bei einer beschränkten Zugriffsdistanz von 131 072 das Verner-Verfahren mit

dem Multi-Core-Tiling auf Skylake und das Verner-Verfahren mit dem Multi-Workgroup-Tiling auf Volta einen Speedup von 1.6 beziehungsweise 1.4 gegenüber der jeweiligen Referenzimplementierung. Dies ist darauf zurückzuführen, dass Skylake 1.21-mal so viel On-Chip-Speicher wie Volta besitzt (43.3 MiByte (L1 + L2 + L3) auf Skylake gegenüber 36 MiByte (Registersatz + L1 + L2) auf Volta), wodurch das Tiling bei größeren Zugriffsdistanzen ähnlich viel Daten über den On-Chip-Speicher wiederverwenden kann. Zudem zeigen die Laufzeitmessungen, dass auf allen untersuchten GPUs und CPUs für kleine Zugriffsdistanzen das trapezoidale Tiling-Schema performanter ist, während für größere Zugriffsdistanzen das hexagonale Tiling-Schema performanter ist. Des Weiteren zeigen die Untersuchungen zur optimalen Tile-Breite und Tile-Höhe, dass sowohl beim Multi-Workgroup-Tiling auf Volta sowie beim Multi-Thread-Tiling auf Skylake bei kleinen Zugriffsdistanzen viele nebenläufige Tiles mit einer geringen Breite und einer großen Höhe eine bessere Performance erzielen, während bei großen Zugriffsdistanzen wenige nebenläufige Tiles mit einer großen Breite und einer geringen Höhe eine bessere Performance erzielen. Bei einer bestimmten Zugriffsdistanz ist jedoch auf Volta ein Tiling mit einer größeren Anzahl von Kernen pro Tile, einer geringeren Höhe und einer größeren Breite performanter als auf Skylake. Die größere Anzahl an Kernen pro Tile können wir wiederum darauf zurückführen, dass Volta weniger On-Chip-Speicher pro GPU-Kern als Skylake pro CPU-Kern besitzt, und somit für eine bestimmte Tile-Größe mehr GPU-Kerne zusammenarbeiten müssen, damit der Workingset der zeitgleich berechneten Tiles in dem On-Chip-Speicher Platz findet.

Das Profiling zeigt, dass bei kleinen Zugriffsdistanzen die Referenzimplementierung, welche wir sowohl auf GPUs als auch auf CPUs nur per Kernelfusion, aber ohne Tiling generiert haben, sowohl auf Volta als auch auf Skylake wie erwartet ein DRAM-Volumen von ungefähr 13.5 GByte benötigt, und zudem stark durch die DRAM-Bandbreite limitiert wird. Ebenso transferiert das Tiling für kleine Zugriffsdistanzen sowohl auf Volta als auch auf Skylake ein ähnlich großes DRAM-Volumen von 1.0 beziehungsweise 1.3 GB. Allerdings lastet das Tiling für kleine Zugriffsdistanzen die DP-Rechenwerke von Volta zu 35 % aus, während es die DP-Peak-Performance von Skylake zu 9 % ausnutzt. Diese geringere Ausnutzung der DP-Peak-Performance können wir auf die nicht vollständige Vektorisierung unserer Implementierung des Tilings für CPUs zurückführen. Ebenso zeigt das Profiling, dass das vom Tiling benötigte DRAM-Volumen auf Volta etwas schneller mit der Größe der Zugriffsdistanz zunimmt als auf Skylake. So benötigt zum Beispiel bei einer beschränkten Zugriffsdistanz von 65 536 das Tiling auf Volta ein DRAM-Volumen von 8.6 GByte, während es auf Skylake ein DRAM-Volumen von 5.9 benötigt. Dies können wir wieder darüber erklären, dass Volta einen etwas kleineren On-Chip-Speicher als Skylake besitzt, wodurch auf Volta bei größeren Zugriffsdistanzen nur der Workingset von weniger großen Tiles, welche wiederum ein größeres Datenvolumen besitzen, in dessen On-Chip-Speicher Platz findet.

Zusammenfassend können wir somit sagen, dass die experimentellen Untersuchungen des Tilings auf GPUs und CPUs ein sehr ähnliches Gesamtbild zeigen, und sich sowohl die Gemeinsamkeiten als auch die Unterschiede größtenteils über die Eckdaten der Architekturen (DP-Peak-Performance, DRAM-Bandbreite, Größe des On-Chip-Speichers, Kernzahl) erklären lassen.

10.5 Fazit

In diesem Kapitel zeigten wir, dass wir über das Tiling optimierte Implementierungsvarianten von expliziten ODE-Verfahren zum Lösen von Anfangswertproblemen mit beschränkter Zugriffsdistanz generieren können. Diese Implementierungsvarianten mit Tiling besitzen gegenüber einer per Kernelfusion generierten Implementierungsvariante, wie wir sie im 9. Kapitel der Kernelfusion herleiteten und untersuchten, eine erhöhte Datenwiederverwendung beziehungsweise ein geringeres Datenvolumen, und dadurch auch ein geringeres DRAM-Volumen und eine verringerte Laufzeit.

Diese Spezialisierung auf Anfangswertprobleme mit beschränkter Zugriffsdistanz war erforderlich, da bei einem allgemeinen Anfangswertproblem ohne beschränkte Zugriffsdistanz das beliebige Zugriffsmuster einer RHS-Operation die Fusion einer LC $\rightarrow$ RHS-Abhängigkeit und damit die Datenwiederverwendung entlang einer solchen LC $\rightarrow$ RHS-Abhängigkeit verhindert. Durch diese Spezialisierung können wir nun jedoch LC $\rightarrow$ RHS-Abhängigkeiten weiterhin nicht direkt zu einem einzigen Kernel verschmelzen, sondern nur über eine partitionierte Fusion zu einem Kernelpaar oder Kerneltriplet. Diese partitionierte Fusion führte nun zu zwei Tiling-Schemata entlang einer RHS $\rightarrow$ LC-Abhängigkeitskette, und zwar zu dem hexagonalen Tiling-Schema und dem trapezoidalen Tiling-Schema. Wir stellten nicht nur vor, wie wir mit diesen beiden Tiling-Schemata ein Tiling über die Stufen, sondern auch ein Tiling über die Zeitschritte realisieren können. Für das Tiling über die Zeitschritte führten wir zusätzlich eine Zeit-Mapping-Distanz, durch welche ein Tile über die Zeitschrittgrenzen verlaufen kann, eine Abroll-Transformation, welche die Datenflussgraphen von mehreren aufeinanderfolgenden Zeitschritten zu einem einzigen Datenflussgraphen kombiniert, eine Prolog-Iteration, um das Tiling am Anfang der Zeitschrittprozedur zu initialisieren, und eine Epilog-Iteration, um das Tiling über die Zeitschritte am Ende der Zeitschrittprozedur abzuschließen, ein. Danach beschrieben wir, wie beim Single-Core-Tiling nicht nur jeder Prozessorkern an seinem Tile arbeiten kann, sondern auch wie beim Multi-Core-Tiling mehrere Kerne an einem Tile zusammenarbeiten können. Durch dieses Multi-Core-Tiling vergrößert sich gegenüber dem Single-Core-Tiling der On-Chip-Speicher, welcher für jedes Tile, das vom Prozessor zeitgleich abgearbeitet wird, zur Verfügung steht. Dadurch passen wiederum größere Tile-Größen noch in den On-Chip-Speicher des Prozessors, wodurch das Multi-Core-Tiling gegenüber dem Single-Core-Tiling für größere Zugriffsdistanzen deutlich effizienter wird.

Ebenso gingen wir darauf ein, wie wir effizienten Code für GPU- und CPU-Kernel mit Tiling erzeugen können, so dass die Kernel effizient die On-Chip-Speicher der jeweiligen Architektur ausnutzen. Dabei versuchen die GPU-Kernels mit Tiling primär Daten über den Registersatz der GPU wiederzuverwenden, während die CPU-Kernel primär versuchen Daten über die Caches der CPU wiederzuverwenden. Damit in unserer Implementierung ein GPU-Kernel mit Tiling Daten über den Registersatz wiederverwenden kann, besteht es aus keinen Schleifen. Allerdings wird über if-Bedingungen sichergestellt, dass ein Workitem eine Vektorgrundoperation nur für die Komponenten des Tiles auswertet und dass es nur die vom Tile benötigten Komponenten aus den permanenten Vektoren lädt oder in die permanenten Vektoren zurückschreibt. Im Gegensatz dazu besteht in unserer Implementierung ein CPU-Kernel mit Tiling aus einer komplexen Schleifenstruktur, welche versucht, die Cache-Hierarchie einer modernen CPU für die Datenwiederverwendung auszunutzen. Falls ein Tile in einem GPU-Kernel nur von einem einzigen Kern abgearbeitet werden soll, müssen wir auf der GPU für jedes Tile eine Workgroup starten (Single-Workgroup-Tiling). Falls dahingegen ein Tile in einem CPU-Kernel nur von einem einzigen Kern abgearbeitet werden soll, reicht es auf CPUs aus, für jeden Kern einen Thread zu

starten und jedem Thread an seinen eigenen Tiles arbeiten zu lassen (Single-Thread-Tiling). Um zu erzielen, dass auf der GPU ein Tile von mehreren Kernen abgearbeitet wird, müssen mehrere Workgroups auf unterschiedlichen GPU-Kernen an einem Tile zusammenarbeiten (Multi-Workgroup-Tiling). Dahingegen können wir auf CPUs die Zusammenarbeit von mehreren Kernen an einem Tile darüber erzielen, indem wir jeweils mehrere Threads an einem Tile zusammenarbeiten lassen (Multi-Thread-Tiling), und diesen Threads unterschiedliche CPU-Kerne zuweisen. Auf beiden Architekturen berücksichtigten wir eine Vielzahl weiterer Architektur-spezifischer Details, um den Code unserer Kernels zu optimieren: Auf GPUs verwirklichen wir einen effizienten Datentransfer für fusionierte LC $\rightarrow$ RHS-Abhängigkeiten beim Single-Workgroup-Tiling über den Scratchpad-Speicher und beim Multi-Workgroup-Tiling über den L2-Cache. Ebenso verwenden wir für effiziente Synchronisation bei fusionierten LC $\rightarrow$ RHS-Abhängigkeiten innerhalb eines Single-Workgroup-Tiles die Hardware-Barrieren eines GPU-Kerns und innerhalb eines Multi-Workgroup-Tiles optimierte Software-Barrieren. Auf CPUs vermeiden wir den Fetch-On-Write-Overhead des Last-Level-Caches durch Non-Temporal-Store-Instruktionen. Zudem vermeiden wir ein unnötiges Zurückschreiben von produzierten Vektorkomponenten in die permanenten Vektoren per Array-Contraction. Zusätzlich realisieren wir auf CPUs das Lesen und Zurückschreiben von Vektorkomponenten in beziehungsweise aus permanenten Vektoren am Rand eines Tiles effizient über zusätzliche Schleifenstrukturen.

Letztlich führten wir experimentelle Untersuchungen auf GPUs und CPUs für die mit dem Tiling erzeugten Implementierungsvarianten durch. Diese Untersuchungen zeigen, dass sich mit dem Tiling gegenüber der Kernelfusion das DRAM-Volumen deutlich reduzieren lässt. Dadurch kann das Tiling gegenüber der Kernelfusion die Performance deutlich steigern. So lässt sich mit einer Implementierungsvariante mit Tiling gegenüber einer Referenzimplementierung nur mit Kernelfusion bei den getesteten Verfahren bei kleinen Zugriffsdistanzen auf den getesteten GPUs ein Speedup zwischen 1.75 und 8.2 und auf den getesteten CPUs ein Speedup zwischen 7.5 und 14.1 erzielen. Ebenso zeigten die Untersuchungen, dass bereits bei kleinen bis mittelgroßen Zugriffsdistanzen das Multi-Core-Tiling gegenüber dem Single-Core-Tiling auf Grund der höheren Performance zu bevorzugen ist. So erzielt zum Beispiel auf der getesteten Volta-GPU beim Verner-Verfahren das Single-Workgroup-Tiling als eine Implementierung des Single-Core-Tilings, nur bis zu einer Zugriffsdistanz von 2 048 einen Speedup gegenüber der Referenzimplementierung, während das Multi-Workgroup-Tiling, als eine Implementierung des Multi-Core-Tilings, bei einer beschränkten Zugriffsdistanz von 131 072 noch einen Speedup von 1.4 erzielt. Ebenso erzielt zum Beispiel auf der getesteten Skylake-CPU beim Verner-Verfahren das Single-Thread-Tiling, als eine Implementierung des Single-Core-Tilings, nur bis zu einer beschränkten Zugriffsdistanz von 32 768 einen Speedup gegenüber der Referenzimplementierung, während das Multi-Thread-Tiling, als eine Implementierung des Multi-Core-Tilings, selbst noch bei einer Zugriffsdistanz von 131 072 einen Speedup von 1.6 erzielt. Für größere Zugriffsdistanzen nimmt sowohl der Speedup des Multi-Workgroup-Tilings auf Volta als auch der Speedup des Multi-Thread-Tiling auf der Skylake-CPU immer weiter ab, da die Tilings das DRAM-Volumen gegenüber der Referenzimplementierung immer weniger reduzieren können. Dadurch werden auch die Tilings durch die DRAM-Bandbreite gebunden. Insgesamt zeigen unsere experimentellen Untersuchungen des Tilings auf GPUs und CPUs für beide Architekturen ein sehr ähnliches Gesamtbild, in welchem sich sowohl die vielen Gemeinsamkeiten als auch die Unterschiede größtenteils über die Eckdaten der Architekturen (DP-Peak-Performance, DRAM-Bandbreite, Größe des On-Chip-Speichers, Kernzahl) erklären lassen.

11 Framework

11.1 Überblick

Wir realisierten die Generierung und die Ausführung der Varianten der Basisimplementierung, der Varianten mit Kernelfusion sowie der Varianten mit Tiling anhand der Datenflussgraphrepräsentation eines ODE-Verfahrens über ein automatisiertes Framework in der Programmiersprache C++. Als ODE-Verfahrensklassen stehen in diesem Framework die RK-Verfahren, die Peer-Verfahren, die PIRK-Verfahren und die Adams-Bashforth-Verfahren zur Verfügung. Allerdings kann ein Benutzer auch leicht neue ODE-Verfahren und neue Klassen von ODE-Verfahren entweder durch Erweiterung des C++-Codes des Frameworks oder über eine DSL innerhalb des Frameworks hinzufügen. Unser Framework unterstützt sowohl GPUs als auch CPUs als optimierte Zielplattformen mit sämtlichen Optimierungen, wie wir sie in den vorherigen Kapiteln dieser Arbeit beschrieben. Jedoch ist es einem Benutzer auch leicht möglich, neue Zielplattformen zum Framework hinzuzufügen. Des Weiteren implementiert unser Framework eine Reihe von hilfreichen Funktionalitäten wie automatisches Speichermanagement, NUMA-Unterstützung und eine Autotuning-Funktionalität für das Tiling.

11.2 Workflow für einen Benutzer des Frameworks

Wenn ein Benutzer mit unserem Framework ein Anfangswertproblem lösen möchte, dann muss er folgende Arbeitsschritte befolgen:

1. **Erzeugung des Datenflussgraphen:** Der Benutzer gibt das zu lösende Anfangswertproblem und das für dessen Lösung zu verwendende ODE-Verfahren an. Das Framework erzeugt daraus den dazugehörigen Datenflussgraphen.

2. **Anwendung der Enabling-Transformationen:** Der Benutzer gibt die gewünschten Enabling-Transformationen an. Das Framework manipuliert den Datenflussgraphen entsprechend den Enabling-Transformationen.

3. **Anwendung der Kernelfusion oder des Tilings:** Der Benutzer gibt die gewünschten Fusionen oder das gewünschte Tiling an. Das Framework erzeugt die Kernels mit den gewünschten Fusionen oder dem gewünschten Tiling und trägt sie in den Datenflussgraphen ein.

4. **Generierung des Codes:** Der Benutzer gibt die gewünschte Zielplattform und die zusätzlichen Kernelkonfigurationen für die Codegenerierung an. Das Framework erzeugt aus dem Datenflussgraphen beziehungsweise aus den darinnen enthaltenen Kernels, dem Anfangswertproblem und den angegebenen Kernelkonfigurationen den Code dieser Kernels und kompiliert ihn.

5. **Ausführung des ODE-Verfahrens:** Der Benutzer gibt die initiale Schrittweite, einen maximalen lokalen Diskretisierungsfehler sowie das Integrationsintervall für das Anfangswertproblem an. Das Framework führt das ODE-Verfahren über die im Datenflussgraphen enthaltenen Kernels aus, um das Anfangswertproblem zu lösen.

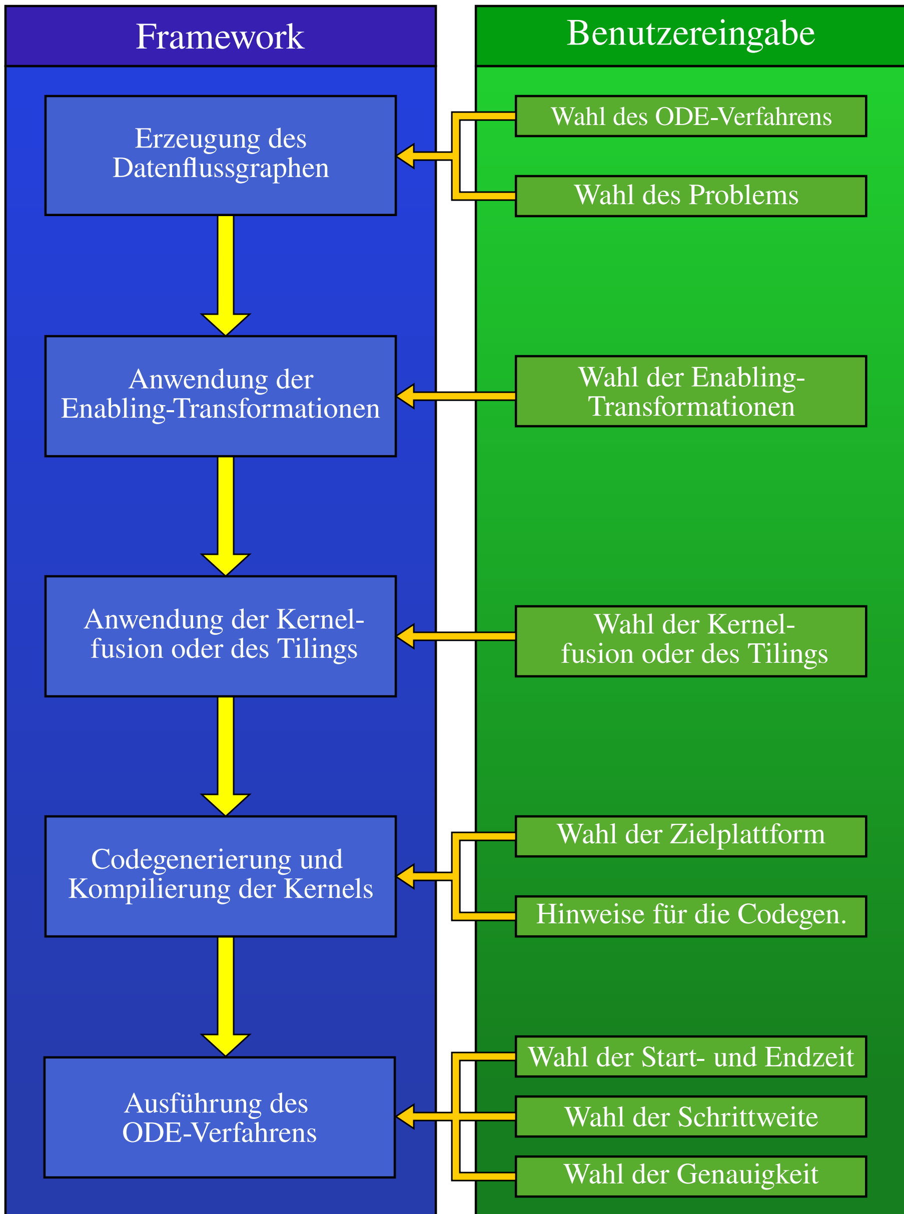


Abbildung 11.1: Schematischer Arbeitsfluss für die Verwendung unseres Frameworks durch einen Benutzer.

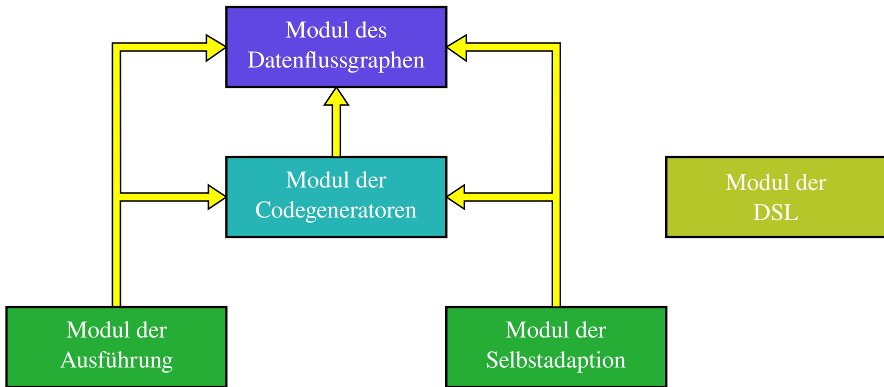


Abbildung 11.2: Module und deren Abhängigkeiten in unserem Framework.

11.3 Gliederung in Module

Unser Framework ist im Wesentlichen in folgende fünf Module gegliedert (siehe Abbildung 11.2):

- **Modul des Datenflussgraphen:** Das *Modul des Datenflussgraphen* beinhaltet die Klassen und Funktionen für den Datenflussgraphen, die Kernels, die Enabling-Transformationen, die Kernelfusion und das Tiling.
- **Modul der Codegeneratoren:** Das *Modul der Codegeneratoren* beinhaltet die Codegeneratoren für die Kernels mit Fusion und Tiling für CPUs und GPUs.
- **Modul der Ausführung:** Das *Modul der Ausführung* dient zur Ausführung eines ODE-Verfahrens anhand seines Datenflussgraphen beziehungsweise der darinnen definierten Kernels.
- **Modul des Autotunings:** Das *Modul des Autotunings* beinhaltet Funktionalitäten, um für einen gegebenen Datenflussgraphen eines ODE-Verfahrens ein optimales Tiling zu finden.
- **Modul der DSL für ODE-Verfahren:** Mit dem *Modul der DSL für ODE-Verfahren* kann ein Benutzer leicht eine neue Verfahrensklasse und ein Verfahren einer Klasse definieren, ohne dafür den C++-Code des Frameworks zu erweitern.

Dabei sind das Modul des Datenflussgraphen und das Modul der DSL für ODE-Verfahren in sich abgeschlossen, während das Modul der Codegeneratoren nur eine Abhängigkeit zu dem Modul des Datenflussgraphen besitzt, aber nicht von dem Modul der Ausführung oder der Selbstadaption abhängig ist. Das Modul der Ausführung und das Modul der Selbstadaption hängen hauptsächlich vom Modul des Datenflussgraphen ab, jedoch besitzen sie auch wenige Abhängigkeiten zum Modul der Codegeneratoren.

11.4 Unterstützte Zielplattformen und Erweiterung um neue Zielplattformen

Für die GPU-Unterstützung verwendet unser Framework die OpenCL-API und CUDA-Driver-API. Für die CPU-Unterstützung erzeugt unser Framework C-Code für die Kernels und lässt diesen C-Code zur Laufzeit zu einer dynamischen Bibliothek kompilieren. Anschließend lädt unser Framework diese dynamische Bibliothek und führt sie aus. Diese Art der CPU-Unterstützung bezeichnen wir im Folgenden als *DLL-Ansatz*. Da dieser DLL-Ansatz vom verwendeten Betriebssystem abhängig ist, implementierten wir für diesen DLL-Ansatz einen Pfad für Windows und Linux. Wieso wir uns für diese Art der CPU-Unterstützung entschieden, werden wir im Abschnitt 13.2 diskutieren.

Allerdings kann ein Benutzer unser Framework auch leicht um eine neue Zielplattform erweitern. Hierfür muss er das Framework wie folgt modifizieren:

- **Keine Modifikationen im Modul des Datenflussgraphen:** Das Modul des Datenflussgraphen ist komplett von der Zielplattform der Kernels unabhängig, und muss deshalb nicht für eine neue Zielplattform modifiziert werden.
- **Implementierung eines neues Codegenerators:** Das Modul der Codegeneratoren benötigt für jede Zielplattform einen eigenen Codegenerator. Jedoch kann sich ein neuer Codegenerator viele Helferfunktionen im Modul des Datenflussgraphen, welche zum Beispiel diejenigen Komponenten berechnen, die ein Tile aus einem permanenten Vektor lesen und in einen permanenten Vektor zurückschreiben muss, oder welche die topologische Reihenfolge der Vektorgrundoperationen innerhalb eines Kernels bestimmen, zu Nutze machen.
- **Geringfügige Erweiterungen im Modul der Ausführung:** Zudem sind kleine Teile des Moduls der Ausführung, und zwar das Starten von Kernels und Teile des automatischen Speichermanagements, von der Zielplattform abhängig, und müssen deshalb von einem Benutzer für die Unterstützung einer neuen Zielplattform um diese erweitert werden.

Somit muss ein Benutzer, um eine neue Zielplattform zu unserem Framework hinzuzufügen, insgesamt nur einen neuen Codegenerator implementieren und die soeben genannten Teile im Modul der Ausführung um eine neue Zielplattform erweitern.

11.5 Unterstützte ODE-Verfahrensklassen

Gegenwärtig unterstützt unser Framework vier ODE-Verfahrensklassen:

- **Explizite RK-Verfahren:** Hier hat ein Benutzer die Auswahl zwischen mehreren vordefinierten expliziten RK-Verfahren, wie das explizite Heun-Euler-Verfahren, das Verner-Verfahren und das Dormant-Prince-Verfahren. Zusätzlich kann ein Benutzer selbst leicht ein neues RK-Verfahren definieren, indem er dessen Butcher-Tableau, also dessen Anzahl an Stufen und dessen Koeffizienten A, $\mathbf{b}$, $\hat{\mathbf{b}}$ und $\mathbf{c}$, angibt. Bei der Konstruktion des Datenflussgraphen für ein RK-Verfahren entfernt unser Framework automatisch diejenigen Argumente einer LC-Operation, die mit einem Gewicht von null in die LC-Operation eingehen.

- **PIRK-Verfahren:** Unser Framework unterstützt PIRK-Verfahren, indem es das Butcher-Tableau des gewünschten impliziten RK-Verfahrens, das als Basisverfahren dienen soll, expandiert, so dass das Butcher-Tableau des PIRK-Verfahrens entsteht. Bei dieser Expansion verschmilzt unser Framework automatisch die redundanten RHS-Operationen des Prädiktorschritts zu einer einzigen RHS-Operation. Über dieses expandierte Butcher-Tableau kann unser Framework das PIRK-Verfahren genauso wie ein explizites RK-Verfahren behandeln. Dadurch kann es die Konstruktionsmethode für den Datenflussgraphen eines RK-Verfahrens benutzen, um den Datenflussgraphen eines PIRK-Verfahrens zu erzeugen.
- **PEER-Verfahren:** Um ein Peer-Verfahren zu erstellen, muss ein Benutzer dessen Stufenzahl und die Verfahrenskoeffizienten A, B und $\mathbf{e}$ angeben.
- **Adams-Bashforth-Verfahren:** Für Adams-Bashforth-Verfahren muss ein Benutzer nur angeben, welches k-Schritt Adams-Bashforth-Verfahren er verwenden möchte.

11.6 Erweiterung des Frameworks um neue ODE-Verfahrensklassen

Auch kann ein Benutzer leicht neue Klassen von ODE-Verfahren zu unserem Framework hinzufügen. Hierfür bietet unser Framework einem Benutzer zwei Möglichkeiten:

- Erweiterung des C++-Codes unseres Frameworks um eine neue Klasse von ODE-Verfahren, sowie
- Spezifikation einer neuen Klasse von ODE-Verfahren in der Klassen-DSL.

Während wir das Hinzufügen einer neuen Klasse von ODE-Verfahren über die Klassen-DSL in Abschnitt 11.11 erläutern, wollen wir im Folgenden kurz das Hinzufügen einer neuen Klasse von ODE-Verfahren über eine Erweiterung des C++-Codes vorstellen. Hierfür muss ein Benutzer unser Framework um Folgendes erweitern:

- **Erweiterung des C++-Codes um eine C++-Konstantenklasse:** Ein Objekt dieser C++-*Konstantenklasse* (siehe Abbildung 11.3 für die Konstantenklasse der RK-Verfahren) soll ein Verfahren der hinzuzufügenden Klasse von ODE-Verfahren über dessen *zeitinvariante Konstanten* repräsentieren. Dafür muss ein jedes Objekt der Konstantenklasse die zeitinvarianten Konstanten eines Verfahrens dieser Klasse von ODE-Verfahren abspeichern, das heißt zum Beispiel bei einem RK-Verfahren die zeitinvarianten Konstanten A, $\mathbf{b}$, $\hat{\mathbf{b}}$ und $\mathbf{c}$. Zusätzlich muss diese Konstantenklasse für ein Interface zum Framework von einer abstrakten Konstantenklasse (`Abstract_Constants`) erben. Diese Konstantenklasse muss zudem folgende Funktionen implementieren:
  - **Konstruktor:** Zunächst sollte ein Benutzer einen Konstruktor für die erbende Konstantenklasse implementieren, welcher die zeitinvarianten Konstanten des ODE-Verfahrens als Argument entgegennimmt und daraus ein Objekt der Konstantenklasse erstellt.
  - **Funktion zur Rückgabe der Zeichenkette einer Konstantenstruktur:** Darüber hinaus ist es erforderlich, dass die erbende Konstantenklasse im Framework eine virtuelle Funktion der abstrakten Konstantenklasse namens

`constant_struct_definition` überschreiben. Diese virtuelle Funktion gibt über eine Zeichenkette die Definition einer C-Struktur, die die zeitvarianten Koeffizienten des Verfahrens abspeichert, zurück. Beispielsweise müsste diese C-Struktur bei RK-Verfahren die Koeffizienten $h_\kappa \cdot A$, $h_\kappa \cdot \mathbf{b}$, $h_\kappa \cdot \hat{\mathbf{b}}$ und $h_\kappa \cdot \mathbf{c}$ beinhalten. Diese Zeichenkette wird dann von dem Codegenerator in den Quelltext der Kernels eingefügt.

  – **Funktion zur Berechnung der zeitvarianten Koeffizienten:** Ebenso muss die erbende Konstantenklasse im Framework die virtuelle Funktion `compute_time_variant_coefficients` der abstrakten Konstantenklasse überschreiben. Diese Funktion berechnet aus den *zeitinvarianten Konstanten* des ODE-Verfahrens, der aktuellen Simulationszeit t_κ und der aktuellen Schrittweite h_κ die *zeitvarianten Koeffizienten* des ODE-Verfahrens. Dabei muss diese Funktion die berechneten zeitvarianten Koeffizienten als einen Vektor von `double`-Werten in derjenigen Reihenfolge sequentialisiert zurückgeben, wie sie in der C-Struktur der zeitvarianten Koeffizienten für die Kernels definiert wurde. Dadurch kann die Ausführung diese Funktion während der Zeitschrittprozedur aufrufen, um die zeitvarianten Koeffizienten des ODE-Verfahrens zu berechnen, und diese Koeffizienten dann an die Kernels zu übergeben.

- **Erweiterung des C++-Codes um eine Konstruktionsmethode für den Datenflussgraphen:** Diese Konstruktionsmethode muss aus einem Objekt der Konstantenklasse, beziehungsweise aus den darinnen enthaltenen zeitinvarianten Konstanten, den Datenflussgraphen des dazugehörigen ODE-Verfahrens konstruieren. Wir werden diese Konstruktionsmethode in Abschnitt 11.7.2 näher erläutern.

11.7 Modul des Datenflussgraphen

11.7.1 Allgemeines

Das *Modul des Datenflussgraphen* beinhaltet die Klassen und Funktionen für den Datenflussgraphen, die Kernels, die Enabling-Transformationen, die Kernelfusion und das Tiling. Für die Vorstellung dieses Moduls fokussieren wir uns in dieser Arbeit auf diejenigen Grundoperationen, welche zur Erläuterung der grundlegenden Funktionsweise des Frameworks am wichtigsten sind, das heißt auf die Konstantengenerierung, RHS-Operationen und LC-Operationen sowie die Abhängigkeiten zwischen diesen. Das Modul des Datenflussgraphen ist wie folgt strukturiert:

- **Klasse des Datenflussgraphen:** Im Modul des Datenflussgraphen existiert eine zentrale C++-Klasse für den Datenflussgraphen, welche wir im folgenden als *Datenflussgraphklasse* bezeichnen. Dabei repräsentieren die Objekte dieser Klasse jeweils den Datenflussgraphen eines ODE-Verfahrens oder eine Implementierungsvariante eines ODE-Verfahrens. In einem solchen Objekt werden deshalb sämtliche Grundoperationen und Kernels, die der Datenflussgraph des ODE-Verfahrens enthalten soll, abgespeichert.

- **Klassen der Grundoperationen:** Das Modul des Datenflussgraphen implementiert die verschiedenen Grundoperationen, also die (RHS | LC | MAP | RED)-Operationen sowie

```
class RK_Constants: public Abstract_Constants
{
   int _stages;

   std::vector<std::vector<double>> _A;
   std::vector<double> _b;
   std::vector<double> _c;

   RK_Constants(
      int stages,
      const std::vector<std::vector<double>>& A,
      const std::vector<double>& b,
      const std::vector<double>& c):
      _stages(stages), _A(A), _b(b) , _c(c)
      {};

   std::_stages constant_struct_definition(
      const std::string& precision)
      const
   {
      return std::string(
         "typedef struct{\n" +
         precision + " hA[" + std::to_string(_stages) + "]"
                        "[" + std::to_string(_stages) + "];\n" +
         precision + " hb[" + std::to_string(_stages) + "];\n" +
         precision + " thc[" + std::to_string(_stages) + "];\n}"+
         "rk_constants;\n\n");
   }

   std::vector<double> compute_time_variant_coefficients(
      double t_k,
      double h_k)
      const
   {
      std::vector<double> data;
      for (int i = 0; i < _stages; i++)
         for (int j = 0; j < _stages; j++)
            data.push_back(h_k * _A[i][j]);

      for (int i = 0; i < _stages; i++)
         data.push_back(h_k * _b[i]);

      for (int i = 0; i < _stages; i++)
         data.push_back(t_k + h_k * _c[i]);

      return data;
   }
}
```

Abbildung 11.3: Konstantenklasse für die RK-Verfahren

die Konstantengenerierung und die Schrittweitensteuerung, als jeweils eine eigene C++-Klasse. Jedes Objekt einer dieser Klassen repräsentiert dabei eine Grundoperation vom entsprechenden Typ und speichert, neben dem Namen dieser Grundoperation, die eingehenden und ausgehenden Abhängigkeiten dieser Grundoperation ab. Dabei ist jede dieser Abhängigkeiten mit einer Zeitschrittdistanz versehen. Ein Objekt einer LC-Operation speichert zusätzlich für jeden Argumentvektor ein Gewicht ab. Dieses Gewicht kann entweder eine zeitinvariante Konstante sein, welche sie als einen `double` Wert abspeichert, oder ein zeitvarianter Koeffizient, das heißt eine Member-Variable der Konstantenstruktur einer Konstantengenerierung, welchen sie als eine Zeichenkette abspeichert. Auch speichert eine RHS-Operation ihren zeitvarianten Koeffizienten, welcher den zeitlichen Offset der RHS-Operation innerhalb eines Zeitschritts repräsentiert, als Zeichenkette ab, wobei diese Zeichenkette wiederum eine Member-Variable der Konstantenstruktur referenziert. Des Weiteren speichert ein jedes Objekt einer Grundoperation den zeitlichen Offset dieser Grundoperation innerhalb eines Zeitschritts ab, während (LC | MAP)-Operationen zusätzlich noch den Typ ihres Ergebnisvektors (Systemzustand, zeitliche Ableitung oder Fehlervektor) als einen Aufzählungstypen abspeichern. Beides dient dazu, dass das Modul der Ausführung für Mehrschrittverfahren die benötigten Startvektoren automatisch initialisieren kann.

- **Klasse der Kernels:** Das Modul des Datenflussgraphen repräsentiert ein jedes Kernel über ein Objekt einer C++-Klasse, eine sogenannte *Kernelklasse*. Dabei speichert ein Objekt dieser Kernelklasse, neben dem Namen des Kernels, Verweise auf diejenigen Vektorgrundoperationen ab, die das Kernel berechnen soll. Dabei ist jeder dieser Verweise mit der Zeit-Mapping-Distanz und dem Argumentintervall des Kernels für diese Vektorgrundoperationen versehen. Für das Tiling unterstützt unser Framework nur ein eindimensionales Argumentintervall, während unser Framework, um bei den Varianten mit Kernelfusion den zweidimensionalen oder dreidimensionalen Iterationsraum von 2D- beziehungsweise 3D-GPU-Kernels zu ermöglichen, auch ein zweidimensionales oder dreidimensionales Argumentintervall unterstützt. Unser Framework definiert jede Dimension für ein Argumentintervall über die Periodenlänge, die Größe des zu berechnenden Blockes in einer Periode sowie den Offset für den zu berechnenden Block innerhalb einer Periode.

- **Interface für ausführbare Objekte:** Des Weiteren implementiert das Modul des Datenflussgraphen das Konzept der ausführbaren Objekte als C++-Interface, welches von der Kernelklasse, der Konstantengenerierung und der Schrittweitensteuerung implementiert wird. Dieses Interface dient dazu, dass das die Ausführung für jeden Zeitschritt einmal in dem Datenflussgraphen über sämtliche ausführbare Objekte in deren topologischen Sortierung iterieren und dabei jedes dieser Objekte ausführen kann.

11.7.2 Konstruktion eines Datenflussgraphen für eine neue Klasse von ODE-Verfahren

Als Nächstes werden wir vorstellen, wie ein Benutzer eine Konstruktionsmethode für die Datenflussgraphen einer neuen Klasse von ODE-Verfahren implementieren kann. Für das Hinzufügen von Grundoperationen zu einem Datenflussgraphen bietet die Datenflussgraphklasse für jeden Typ von Grundoperation eine entsprechende Funktion an, die die Grundoperation mit einem gegebenen Namen und Zeitoffset erstellt und hinzufügt (siehe Abbildung 11.4). Ebenso

implementiert jede C++-Klasse einer Grundoperation für jedes ihrer Argumente eine Funktion, die eine andere Grundoperation als dieses Argument mit einer gegebenen Zeitschrittdistanz setzt. Dabei fügt diese Funktion beim Setzen eines Arguments auch eine Abhängigkeit zwischen diesen beiden Grundoperation im Datenflussgraphen hinzu (siehe Abbildung 11.5). Die Abbildung 11.6 zeigt beispielsweise die Konstruktionsmethode für den Datenflussgraphen eines Adams-Bashforth-Verfahrens, und verdeutlicht, dass sich ein solcher Datenflussgraph in unserem Framework mit nur wenigen Codezeilen generieren lässt.

11.7.3 Anwendung der Enabling-Transformationen, und Erstellung der Varianten

Um die Enabling-Transformationen auf den Datenflussgraphen anzuwenden, beziehungsweise um eine Variante der Basisimplementierung oder um eine Variante mit Fusion oder Tiling zu erzeugen, stellt das Modul des Datenflussgraphen einem Benutzer folgende Möglichkeiten zur Verfügung:

- **Anwendung der Enabling-Transformationen:** Das Modul des Datenflussgraphen implementiert sämtliche in dieser Arbeit vorgestellten Enabling-Transformationen, das heißt die Cloning-Transformation, die Splitting-Transformation, die Move-Transformation und die Abroll-Transformation. Ein Benutzer kann diese Enabling-Transformationen jeweils mit einem einzigen Funktionsaufruf auf einen Datenflussgraphen beziehungsweise auf eine Grundoperation anwenden.
- **Erstellung der Basisimplementierung:** Ein Benutzer kann aus dem Datenflussgraphen eines ODE-Verfahrens mit einem einzigen Funktionsaufruf die Basisimplementierung erstellen. Dabei erstellt dieser Funktionsaufruf für jeweils jede Grundoperation im Datenflussgraphen ein Kernel.
- **Generierung einer Variante mit Kernelfusion:** Für die Generierung einer Variante mit Kernelfusion aus einem Datenflussgraphen stellt das Modul des Datenflussgraphen einem Benutzer zwei Möglichkeiten zur Verfügung. Bei der ersten Möglichkeit kann ein Benutzer selbst über einen Funktionsaufruf mehrere Grundoperationen angeben, die zu einem Kernel fusioniert werden sollen. Hierbei ist der Benutzer selbst dafür verantwortlich, dass jede Vektorgrundoperation von einem Kernel berechnet wird, sowie dass durch die Kernelfusion keine zyklischen Abhängigkeiten zwischen den Kernels entstehen. Bei der zweiten Möglichkeit kann ein Benutzer auch mit einem einzigen Funktionsaufruf entlang der gesamten RHS $\rightarrow$ LC-Abhängigkeitskette des ODE-Verfahrens jeweils sämtliche Vektorgrundoperationen in einem RHS $\rightarrow$ LC-Glied zu einem Kernel verschmelzen lassen. Bei beiden Möglichkeiten muss der Benutzer für jedes Kernel zudem ein Argumentintervall spezifizieren.
- **Generierung einer Variante mit Tiling:** Für die Generierung einer Variante mit Tiling aus einem Datenflussgraphen stellt das Modul des Datenflussgraphen einem Benutzer wieder zwei Möglichkeiten zur Verfügung. Bei der ersten Möglichkeit kann der Benutzer selbst eine Folge von RHS $\rightarrow$ LC-Gliedern definieren, entlang der das Framework das Tiling generieren soll. Bei der zweiten Möglichkeit generiert das Framework das Tiling entlang der gesamten RHS $\rightarrow$ LC-Abhängigkeitskette des ODE-Verfahrens. Sowohl bei dem trapezoidalen Tiling-Schema als auch bei dem hexagonalen Tiling-Schema kann der

```
Constant_Generation& Method_Graph::add_constant_generation(
      const std::string& name)
LC_Operation& Method_Graph::add_lc_operation(
      const std::string& name, double dt_rel, Vectype type)
RHS_Operation& Method_Graph::add_rhs_operation(
      const std::string& name, double dt_rel)
```

Abbildung 11.4: Funktionen für das Hinzufügen von Grundoperationen zu einem Datenflussgraphen. Bei diesen Funktionen wird das Argument `dt_rel` als zeitlicher Offset der hinzugefügten Grundoperation innerhalb eines Zeitschritts interpretiert. Ebenso muss beim Hinzufügen einer (LC | MAP)-Operation per Enum angegeben werden, welchen Typ von Vektor (Systemzustand, zeitliche Ableitung oder Fehlervektor) die Grundoperation erzeugt.

```
void RHS_Operation::set_argument_constants(Constant_Generation& gen
         int timestep_distance, const std::string& coefficient_name)
void RHS_Operation::set_argument_vector(Vector_Operation& argument,
         int timestep_distance)

void LC_Operation::set_argument_constants(Constant_Generation& gen
         int timestep_distance, const std::string& coefficient_name)
void LC_Operation::add_argument_vector(Vector_Operation& argument,
         double weight, int timestep_distance)
void LC_Operation::add_argument_vector(Vector_Operation& argument,
         const std::string& coefficient_name, int timestep_distance)
```

Abbildung 11.5: Funktionen für das Hinzufügen von Abhängigkeiten zwischen Grundoperationen. Bei diesen Funktionen repräsentiert die Zeichenkette `coefficient_name` diejenige Membervariable in der Konstantenstruktur, welche den entsprechenden zeitvarianten Koeffizienten für die jeweilige Grundoperation abspeichert, und der Integer `timestep_distance` die Zeitschrittdistanz der Abhängigkeit.

```
Graph::Graph(const AB_Constants & ab_constants, int access_dist):
_access_dist(access_dist)
{
   _ode_constants = std::make_unique<AB_Constants>(ab_constants);

   Constant_Generation& constant_generation =
      add_constant_generation("ab", 0.0);
   RHS_Operation& f = add_rhs_operation("f", 0.0);
   LC_Operation& y = add_lc_operation("y", 1.0, Vectype::y);

   f.set_argument_vector(y, 1);
   y.set_argument_constants(constant_generation, 0);
   y.add_argument_vector(y, 1.0, 0);
   for (int i = 0; i < ab_constants.previous_steps(); i++)
      if (ab_constants.b[i] != 0.0)
         y.add_argument_vector(f, "b[" + to_string(i) + "]", i);
}
```

Abbildung 11.6: Konstruktionsmethode für den Datenflussgraph für ein Adams-Bashforth-Verfahren.

Benutzer im Framework jede dieser beiden Möglichkeiten jeweils über einen einzigen Funktionsaufruf umsetzen. Für das trapezoidale Tiling-Schema muss ein Benutzer bei beiden Möglichkeiten zusätzlich noch die Höhen der Zeilen aus Trapezoiden und für jede Zeile jeweils noch die Breiten der geraden und ungeraden Trapezoide angeben. Für das hexagonale Tiling-Schema muss ein Benutzer bei beiden Möglichkeiten die Höhen der Hexagone sowie die Breite der ungeraden Spalten aus Hexagonen und die Breite der geraden Spalten aus Hexagonen angeben. Beide Tiling-Schemata können in unserem Framework dafür verwendet werden, um sowohl ein Tiling über die Stufen als auch ein Tiling über die Zeitschritte zu realisieren. Zudem generiert unser Framework bei einem Tiling über die Zeitschritte automatisch die Prologkernels und die Epilogkernels.

11.7.4 Hilfsfunktionen für die Codegenerierung und die Ausführung

Das Modul des Datenflussgraphen implementiert eine Vielzahl von Funktionalitäten, die für das Modul der Codegeneratoren und für das Modul der Ausführung sehr hilfreich sind. Die wichtigsten hiervon sind:

- die Bestimmung der LC $\rightarrow$ RHS-Abhängigkeitskette im Datenflussgraphen,
- die topologische Sortierung der ausführbaren Objekte im Datenflussgraphen,
- die topologische Sortierung der Vektorgrundoperationen in einem Kernel,
- die Bestimmung der von einem Kernel berechneten LC $\rightarrow$ RHS-Glieder und die Vektorgrundoperationen in jedem dieser Glieder,
- die Bestimmung der Argumente eines Kernels, das heißt, welche Arrays mit den zeitvarianten Koeffizienten der Konstantengenerierung das Kernel benötigt, sowie aus welchen permanenten Vektoren das Kernel liest und in welche permanenten Vektoren das Kernel schreibt,
- die Berechnung derjenigen Intervalle an Komponenten für einen Vektor, die ein Kernel mit Tiling aus einem permanenten Vektor und damit aus dem DRAM laden muss, sowie
- die Berechnung derjenigen Intervalle an Komponenten für eine Vektorgrundoperation, die ein Kernel mit Tiling in deren permanenten Vektor und damit den DRAM zurückschreiben muss.

11.8 Modul der Codegeneratoren

Das *Modul der Codegeneratoren* besteht aus zwei voneinander unabhängigen Teilmodulen, und zwar einen Codegenerator für GPU-Kernels und einen Codegenerator für CPU-Kernels. Beide Codegeneratoren nehmen ein Objekt der Kernelklasse aus dem Modul des Datenflussgraphen entgegen, um daraus ausführbaren Code zu erzeugen. Bei diesem Objekt kann es sich um ein Kernel der Basisimplementierung, um ein fusioniertes Kernel oder um ein Kernel mit Tiling handeln. Der von den Codegeneratoren erzeugte Code besitzt sämtliche Speicheroptimierungen, wie wir sie in den vorherigen Kapiteln vorstellten. Zudem können die Codegeneratoren für ihre Codegenerierung auf viele Hilfsfunktionen aus dem Modul des Datenflussgraphen zugreifen. Im Detail implementierten wir die beiden Codegeneratoren für GPU-Kernels und CPU-Kernels wie folgt:

Codegenerator für GPU-Kernels: Dieser Codegenerator erzeugt für ein gegebenes Kernel zunächst einen CUDA-C- oder einen OpenCL-C-Quelltext. Dabei ist der Codegenerator vom Benutzer durch viele Parameter konfigurierbar. Insbesondere kann der Benutzer für alle Kernels angeben, wie viele Blöcke aus Vektorkomponenten ein Thread berechnen und wie groß ein solcher Block sein soll. Zusätzlich kann ein Benutzer bei Kernels mit Tiling angeben, ob der Codegenerator ein Kernel mit Single- oder Multi-Workgroup-Tiling generieren soll und welcher Stride zwischen den Blöcken eines Threads im Kernel bestehen soll. Bei der Erzeugung des Quelltexts für CUDA-Kernels und OpenCL-Kernels nutzt der GPU-Codegenerator aus, dass CUDA-C und OpenCL-C beide C-Dialekte und damit sehr ähnlich sind. Dadurch kann er für beide Sprachen denselben Generierungspfad für den Code des Kernels verwenden, ohne dabei viele Fallunterscheidungen für die verwendete Sprache zu benötigen. Die meisten Fallunterscheidungen, die dennoch bestehen bleiben, vermeidet unser GPU-Codegenerator, indem er selbst für OpenCL weitestgehend CUDA-C generiert, zusätzlich aber bei OpenCL am Anfang des Quelltexts eine Headerdatei einfügt, die per Präprozessoranweisungen jeden CUDA-spezifischen Bezeichner in das jeweiligen OpenCL-Analogon umdefiniert. Nach der Generierung des Quelltextes des Kernels ruft der Codegenerator den Laufzeitcompiler der jeweiligen GPU-API auf, das heißt bei CUDA die Funktion `nvrtcCompileProgram` des *NVRTC-Compilers* von NVIDIA [W20] und bei OpenCL die Funktion `clBuildProgram`, welche den GPU-Compiler der jeweiligen OpenCL-Plattform aufruft. Abschließend gibt der GPU-Codegenerator das so erzeugte GPU-API-Kernel zurück.

Codegenerator für CPU-Kernels: Dieser Codegenerator erzeugt für ein gegebenes Kernel zunächst einen C-Quelltext. Dabei ist der Codegenerator vom Benutzer konfigurierbar, das heißt, der Benutzer hat die Wahl zwischen den von uns implementierten Schleifenstrukturen der jeweiligen Variante und beim Tiling die Wahl zwischen Single-Thread-Tiling oder Multi-Thread-Tiling. Zusätzlich kann der Benutzer weitere diverse Performance-Parameter wie die Blockgrößen oder die Subblockgrößen angeben. Anschließend speichert der Codegenerator den Quelltext des jeweiligen Kernels in einer C-Datei ab. Danach startet das Codegenerator über das Betriebssystem einen Compiler-Prozess und befiehlt diesem Compiler-Prozess die C-Datei des Kernels zu einer dynamischen Bibliothek zu kompilieren. Hierfür verwendet der Codegenerator sowohl unter Linux als auch unter Windows die Funktion `system`, welcher er die Kommandozeilenparameter des Compilers übergibt. Als C-Compiler stehen in unserem Framework der *MSVC-Compiler* von Microsoft [W18], der quelloffene *GCC-Compiler* [W19] und der *ICC-Compiler* von Intel [W17] zur Auswahl. Ein Benutzer kann allerdings leicht weitere C-Compiler hinzufügen, da er hierfür nur die Kommandozeilenparameter für den Compiler in unserem Framework eintragen muss. Zuletzt lädt der Codegenerator die dynamische Bibliothek mit der Funktion `dlopen` unter Linux oder der Funktion `LoadLibrary` unter Windows und gibt die geladene Bibliothek zurück.

11.9 Modul der Ausführung

11.9.1 Ablauf der Ausführung

In diesem Abschnitt werden wir den Ablauf der Ausführung eines ODE-Verfahrens durch das *Modul der Ausführung* im Detail vorstellen. Eine Übersicht dieses Ablaufs der Ausführung wird in Abbildung 11.7 gezeigt. Startet ein Benutzer des Frameworks die Ausführung, so initialisiert sie zunächst die initialen permanenten Vektoren des ODE-Verfahrens. Falls es sich bei dem auszuführenden ODE-Verfahren um ein Einschrittverfahren handelt, so kopiert die Aus-

führung lediglich den initialen Systemzustand in einen permanenten Vektor. Falls es sich um ein Mehrschrittverfahren handelt, so benötigt die Ausführung zusätzlich weitere initiale permanente Vektoren. Ein Benutzer kann diese zusätzlichen initialen permanenten Vektoren durch die Ausführung automatisch über den Datenflussgraphen eines Einschrittverfahrens seiner Wahl berechnen lassen kann. Alternativ kann ein Benutzer diese initialen permanenten Vektoren auch direkt bei der Initialisierung der Ausführung angeben.

Als Nächstes führt die Ausführung die Zeitschrittprozedur via einer Zeitschleife aus. Jede Iteration der Zeitschleife besteht aus der Ausführung der ausführbaren Objekte im Datenflussgraphen, das heißt der Konstantengenerierungen, der Kernels und der Schrittweitensteuerung, in ihrer topologischen Sortierung:

- **Ausführung einer Konstantengenerierung:** Hierfür ruft die Ausführung auf dem Konstantenobjekt die Funktion `compute_time_variant_coefficients` auf, welche aus den zeitinvarianten Konstanten des ODE-Verfahrens dessen zeitvariante Koeffizienten berechnet. Die Ausführung speichert diese zeitvarianten Koeffizienten ab, und übergibt sie an alle Kernels, die sie für die Berechnung ihrer Vektorgrundoperationen benötigen.
- **Ausführung eines Kernels:** Um ein Kernel auszuführen, alloziert die Ausführung zunächst diejenigen permanenten Vektoren, die bislang noch nicht alloziert wurden und in die das auszuführende Kernel schreiben wird. Danach konstruiert die Ausführung die Argumente des Kernels. Anschließend startet sie das Kernel mit den soeben konstruierten Argumenten. Nach dem Starten des Kernels gibt die Ausführung diejenigen permanenten Vektoren, die von keinem nachfolgenden Kernel mehr gelesen werden und die auch nicht als Endergebnis benötigt werden, wieder frei.
- **Ausführung der Schrittweitensteuerung:** Bei der Schrittweitensteuerung überprüft die Ausführung zunächst, ob der lokale Diskretisierungsfehler kleiner als der benutzerdefinierte maximale Diskretisierungsfehler ist. Falls ja, dann geht die Ausführung zum nächsten Zeitschritt über, das heißt sie inkrementiert die Simulationszeit um die aktuelle Schrittweite. Ansonsten wiederholt sie den aktuellen Zeitschritt. In beiden Fällen berechnet die Ausführung für die Schrittweitensteuerung noch aus dem lokalen Diskretisierungsfehler eine neue Schrittweite, mit welcher der nächste Zeitschritt auszuführen beziehungsweise der aktuelle Zeitschritt zu wiederholen ist. Zusätzlich gibt die Ausführung nach der Schrittweitensteuerung die nicht mehr benötigten permanenten Vektoren wieder frei.

Ist keine Schrittweitensteuerung vorhanden, so inkrementiert die Ausführung am Ende eines Zeitschritts lediglich die Simulationszeit um die Schrittweite und gibt ebenfalls die nicht mehr benötigten permanenten Vektoren wieder frei. Des Weiteren, falls die Ausführung eine Variante mit einer Prolog- oder Epilogiteration ausführen soll, so führt sie automatisch die Prologiteration mit den Prologkernels anstelle des ersten Zeitschritts und die Epilogiteration mit den Epilogkernels nach dem letzten Zeitschritt aus.

11.9.2 Permanente Vektoren und deren Speichermanagement

In jedem permanenten Vektor speichert das Modul der Ausführung diejenige (RHS | LC | MAP)-Operation, die diesen Vektor produziert hat, und zusätzlich die Nummer desjenigen Zeitschritts,

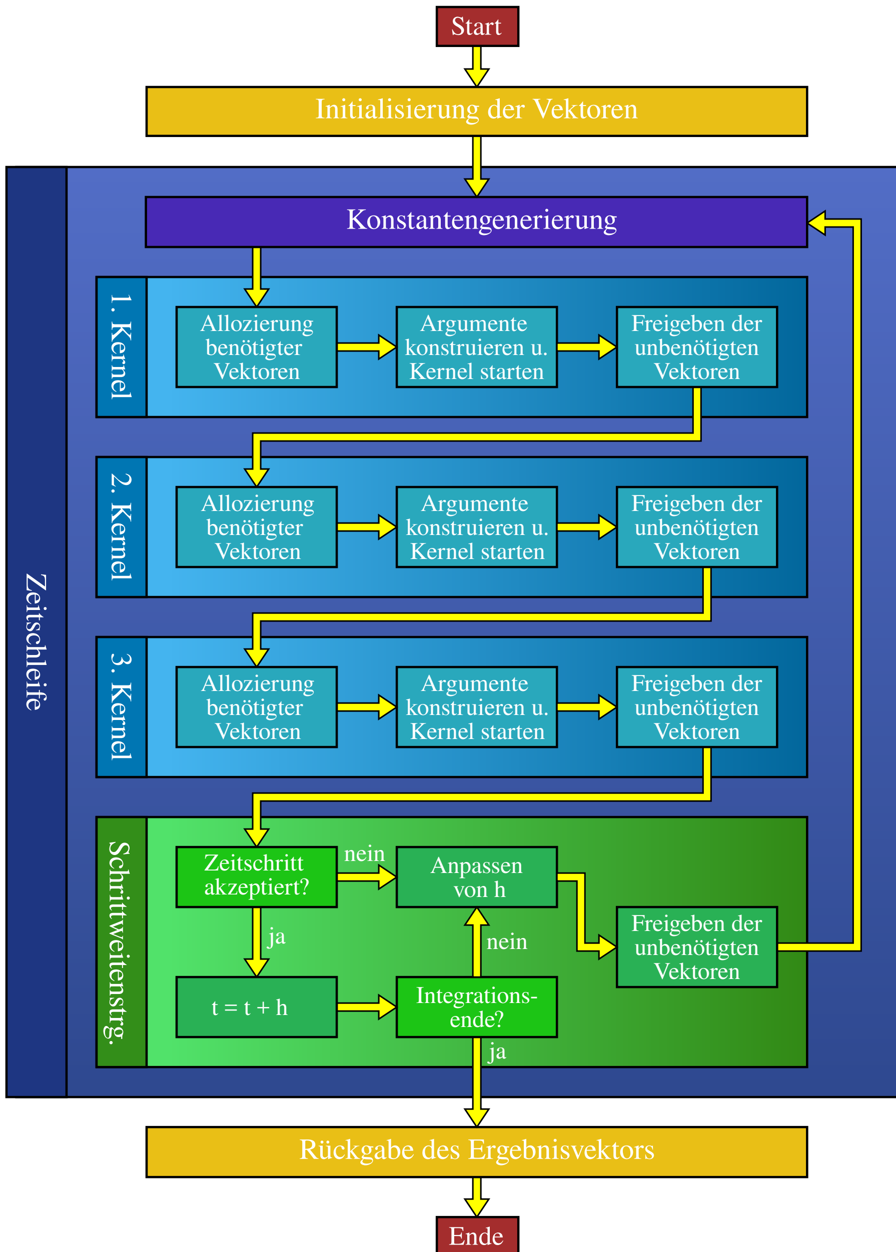


Abbildung 11.7: Vorgehen der Ausführung unseres Frameworks.

in welchem der Vektor produziert wurde, ab. Des Weiteren beinhaltet ein permanenter Vektor bei einer Kernelausführung auf einer GPU einen Verweis auf seinen allozierten Heap-Speicher im DRAM der GPU, während er bei einer Kernelausführung auf einer CPU einen Zeiger auf seinen allozierten dynamischen Speicher im DRAM der CPU beinhaltet.

Das Modul der Ausführung alloziert einen permanenten Vektor automatisch, wenn ein Kernel ausgeführt werden soll, das in einen noch nicht allozierten permanenten Vektor hineinschreibt. Die Regeln für das Freigeben eines Vektors unterscheiden sich, je nachdem, ob das Modul der Ausführung ein ODE-Verfahren mit oder ohne Schrittweitensteuerung ausführt:

- **Freigeben für ein Verfahren ohne Schrittweitensteuerung:** Bei einem Verfahren ohne Schrittweitensteuerung darf ein permanenter Vektor wieder freigegeben werden, wenn im aktuellen Zeitschritt sämtliche Kernels ausgeführt wurden, die diesen Vektor auslesen, und auch in einem zukünftigen Zeitschritt kein Kernel existiert, das diesen Vektor ausliest.
- **Freigeben für ein Verfahren mit Schrittweitensteuerung:** Da bei einem Verfahren mit Schrittweitensteuerung ein Zeitschritt verworfen werden kann, muss für die Freigabe eines permanenten Vektors unterschieden werden, ob dieser permanente Vektor in dem aktuellen oder in einem vergangenen Zeitschritt produziert wurde:
  - **1. Fall: Der permanente Vektor wurde im aktuellen Zeitschritt produziert.** Ein solcher Vektor darf wieder freigegeben werden, falls die Ausführung sämtliche Kernels im aktuellen Zeitschritt ausgeführt hat, die diesen Vektor auslesen, und auch in einem zukünftigen Zeitschritt kein Kernel existiert, das diesen Vektor ausliest. Ein solcher Vektor muss zudem, nachdem die Schrittweitensteuerung den aktuellen Zeitschritt verworfen hat, ebenso verworfen werden.
  - **2. Fall: Der permanente Vektor wurde in einem vergangenen Zeitschritt produziert.** Ein solcher Vektor darf wieder freigegeben werden, wenn die Schrittweitensteuerung den aktuellen Zeitschritt akzeptiert hat, und auch in einem zukünftigen Zeitschritt kein Kernel existiert, das diesen Vektor ausliest.

Um das Allozieren und das Freigeben von Vektoren zu beschleunigen, verwendet unser Framework einen Speicherpool, welcher eine Menge an Allozierungen im Heap-Speicher der GPU beziehungsweise im dynamischen Speicher der CPU beinhaltet.

11.9.3 Starten von Kernels

Das Starten von Kernels unterscheidet sich im Modul der Ausführung, je nachdem welche Plattform verwendet werden soll:

- **CUDA:** Um ein CUDA-Kernel zu starten, muss der Main-Thread der Ausführung zunächst Zeiger auf die Argumente des Kernels in einem Array abspeichern. Anschließend kann er das Kernel mit einem Aufruf der Funktion `cuLaunchKernel` in eine Task-Queue auf der GPU einreihen. Bei diesem Funktionsaufruf muss der Main-Thread unseres Frameworks als ein Argument das Array mit den Zeigern auf die Kernelargumente übergeben, welches die GPU dann an jedes Workitem des Kernels weiterleitet. Als weitere Argumente muss er bei diesem Funktionsaufruf die Dimensionalität des Kernels

und die Anzahl an Workgroups sowie die Anzahl an Workitems pro Workgroup übergeben. Danach kann der Main-Thread auf der CPU mit dem Starten des nächsten Kernels fortfahren, während hierzu asynchron die Kerne der GPU das soeben eingereihte Kernel abarbeiten. Die Ausführung muss den Main-Thread auf der CPU und die GPU erst miteinander synchronisieren, wenn sie das Ergebnis einer Reduktion zurück zur CPU kopieren muss, damit der Main-Thread die Schrittweitensteuerung ausführen kann. Dadurch vermeidet die Ausführung den Overhead einer häufigen Synchronisation zwischen GPU und CPU.

- **OpenCL:** Das Starten von Kernels mit OpenCL ist weitestgehend analog zu CUDA, das heißt der Main-Thread der Ausführung muss das Kernel über einen Aufruf der Funktion `clEnqueueNDRangeKernel` mit einer gegebenen Anzahl von Workgroups und Workitems per Workgroup in eine Task-Queue auf der GPU einreihen. Allerdings müssen in OpenCL vor dem Einreihen des Kernels in eine Task-Queue die Argumente des Kernels mit der Funktion `clSetKernelArg` an das Kernel gebunden werden.
- **CPU:** Um ein Kernel auf der CPU mit dem DLL-Ansatz auszuführen, implementiert das Modul der Ausführung einen Pool an Worker-Threads, die es zu Beginn einmalig initialisiert. Um das Kernel zu starten, setzt der Main-Thread der Ausführung den Zähler des Kernels für die Anzahl an ausgeführten Blöcken auf null zurück und konstruiert ein Array von Zeigern auf die Argumente des Kernels. Anschließend benachrichtigt der Main-Thread die Workerthreads, dass sie nun dieses Kernel ausführen sollen. Die Workerthreads rufen nun die Kernelfunktion aus der dynamisch geladenen Bibliothek über deren Funktionszeiger mit dem Array von Zeigern als Argument auf. Nachdem die Workerthreads sämtliche Blöcke ausgeführt haben, benachrichtigen sie den Main-Thread, dass sie fertig sind, so dass der Main-Thread mit der Zeitschrittprozedur fortfahren kann.

11.9.4 Unterstützung für NUMA-CPUs

Das Modul der Ausführung implementiert ebenfalls eine Unterstützung von *Non-Uniform-Memory-Access-CPUs* (*NUMA-CPUs*). Die Thematik von NUMA wird zum Beispiel in der Arbeit [Z43] ausgiebig behandelt. Für diese NUMA-Unterstützung führt das Modul der Ausführung auf NUMA-CPUs eine blockweise Aufteilung von den Systemkomponenten beziehungsweise von den Komponenten der permanenten Vektoren auf die NUMA-Domains der NUMA-CPU durch (siehe Abbildung 11.8). Diese Aufteilung erzielt das Modul der Ausführung bei der Allozierung eines permanenten Vektors über die *First-Touch-Policy* des Betriebssystems. Hierfür lässt das Modul der Ausführung, nachdem es über einen Aufruf von `malloc` den Speicherplatz für einen permanenten Vektor alloziert hat, jede NUMA-Domain einmal über sämtliche Komponenten ihres Blockes in diesem permanenten Vektor iterieren, so dass das Betriebssystem die Speicherseiten eines Blockes in dem DRAM der passenden NUMA-Domain alloziert. Für die Ausführung eines Kernels implementiert das Modul der Ausführung für jede NUMA-Domain einen Thread-Pool mit jeweils einem atomaren Zähler für die Blöcke beziehungsweise die Tiles des Kernels. Bei der Ausführung eines Kernels berechnet jeder Thread-Pool nur diejenigen Blöcke beziehungsweise diejenigen Tiles, welche sich innerhalb des Blockes seiner NUMA-Domain befinden (siehe Abbildung 11.9). Auf diese Weise reduziert das Modul der Ausführung die Anzahl an Remote-Speicherzugriffen, das heißt die Zugriffe von einer NUMA-Domain auf den Speicher einer anderen NUMA-Domain, erheblich.

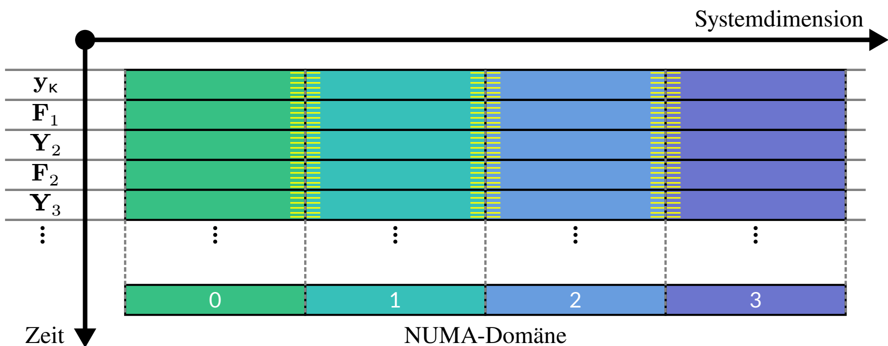


Abbildung 11.8: Blockweise Aufteilung der Vektorkomponenten auf die NUMA-Domains einer NUMA-CPU. Die in der Abbildung gelb schattierten Bereiche am Rand jedes Blocks einer NUMA-Domain symbolisieren diejenigen Vektorkomponenten, welche die benachbarte NUMA-Domain über Remote-Zugriffe lesen oder schreiben muss.

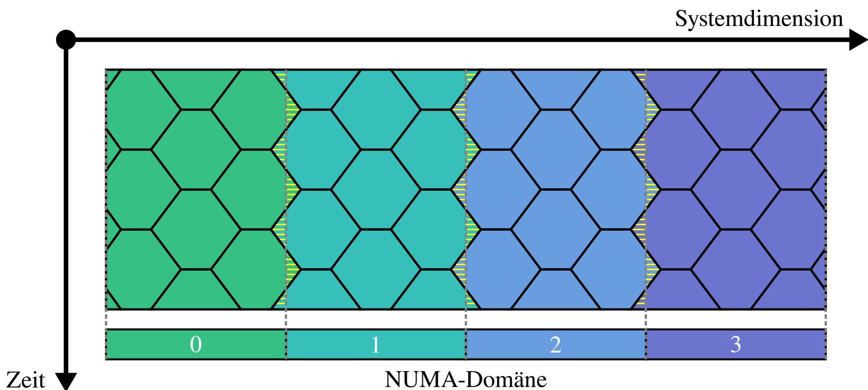


Abbildung 11.9: Aufteilung der Tiles auf die NUMA-Domains einer NUMA-CPU. Die in der Abbildung gelb schattierten Bereiche am Rand der rechtesten und linkesten Tiles einer NUMA-Domain symbolisieren diejenigen Vektorkomponenten in dem entsprechenden Tile, welche das Tile über Remote-Zugriffe zu der benachbarten NUMA-Domain lesen oder schreiben muss.

11.10 Modul des Autotunings

Das Modul des Autotunings ist in der aktuellen Version des Frameworks nur rudimentär vorhanden. Es unterstützt deshalb nur das Autotuning für das Tiling, wie wir es in Abschnitt 10.2.11 beschrieben. Ebenso unterstützt es aktuell nur GPUs als Zielarchitektur. Allerdings lässt sich mit Hilfe dieses Moduls mit nur einem Funktionsaufruf das beste Tiling und die besten Kernelkonfigurationen für dieses Tiling bestimmen. Zusätzlich kann ein Benutzer den durch das Autotuning durchsuchten Konfigurationsraum leicht einschränken, wodurch zum Beispiel das

Autotuning seine Suche nur entweder auf das hexagonale oder trapezoidale Tiling sowie entweder auf das Single-Workgroup-Tiling oder auf das Multi-Workgroup-Tiling beschränkt. Dieses Autotuning erfolgt in einem Offline-Ansatz, das heißt, die Kernel führen ihre Berechnungen auf einer Menge von benutzerdefinierten permanenten Vektoren aus, ohne dabei das Anfangswertproblem zu lösen. Somit eignet sich diese Art des Autotunings nicht für Kernel, bei welchen der Kontrollfluss stark von den Daten in den permanenten Vektoren abhängig ist. Zusätzlich, da unser Ansatz mit Offline-Autotuning seine Ergebnisse verwirft und nicht zur Lösung des Anfangswertproblems verwendet, erzeugt er einen größeren Overhead als ein Ansatz mit Online-Autotuning. Das Modul des Autotunings soll aber in zukünftigen Versionen des Frameworks deutlich erweitert werden, so dass es die Enabling-Transformationen und die Kernelfusion automatisch anwendet, um optimierte Implementierungsvarianten für Anfangswertprobleme mit beschränkter Zugriffsdistanzen zu finden, CPUs als Zielplattform unterstützt, und zudem über optimierte Suchalgorithmen und Performance-Modelle schneller optimierte Varianten mit Tiling erzeugen kann.

11.11 Modul der DSL für ODE-Verfahren

Im Rahmen dieser Arbeit entwickelten wir eine DSL, welche zur Definition einer Klasse von ODE-Verfahren und von einem spezifischen ODE-Verfahren einer zuvor definierten Klasse dient. Dementsprechend gliedert sich die DSL in eine *Klassen-DSL* und einer *Verfahren-DSL*.

11.11.1 Klassen-DSL

Die *Klassen-DSL* (siehe Abbildung 11.10) dient zur Definition einer Klasse von ODE-Verfahren. Damit die Beschreibung einer Klasse von ODE-Verfahren mit der Klassen-DSL möglichst analog zu der mathematischen Definition einer Klasse von ODE-Verfahren erfolgen kann, entwarfen wir die Klassen-DSL als *Datenflussprogrammiersprache* (für eine ausgiebige Behandlung der Thematik der Datenflussprogrammiersprachen siehe zum Beispiel [Z44]). Für die Definition einer Klasse von ODE-Verfahren benötigt diese Klassen-DSL nur ein Dokument, welches noch einmal in mehrere Blöcke gegliedert ist:

Meta-Block: Der *Meta-Block* definiert, welche Metainformationen ein ODE-Verfahren der definierten Klasse über sich angeben kann, wie zum Beispiel seinen Namen, seine Stufenzahl oder seine Ordnung.

Constants-Block: Der *Constants-Block* deklariert diejenigen zeitinvarianten Konstanten (Skalare oder ein- oder mehrdimensionale Arrays von Skalaren), die die Schrittfunktion eines ODE-Verfahrens der definierten Klasse für einen Zeitschritt benötigt. Als Array-Größen können die zuvor definierten skalaren Konstanten im Constants-Block oder im Meta-Block dienen.

Vectors-Block: Der *Vectors-Block* deklariert diejenigen Vektoren, die die Schrittfunktion eines ODE-Verfahrens der definierten Klasse für einen Zeitschritt benötigt. Bei der Deklaration muss für jeden Vektor zusätzlich ein Typ (Systemzustand, zeitliche Ableitung oder Fehlervektor) angegeben werden, damit die Ausführung diese Vektoren später initialisieren kann. Des Weiteren kann dieser Block auch Arrays von Vektoren deklarieren. Als Größen für diese Arrays können wieder die zuvor definierten Konstanten im Constants-Block oder Meta-Block dienen.

Stepfunction-Block: Der *Stepfunction-Block* beschreibt die Schrittfunktion eines ODE-Verfahrens der definierten Klasse von ODE-Verfahren und verwendet für diese Beschreibung den Ansatz der Datenflussprogrammierung. Aus diesem Grund erfolgt die Definition der

Schrittfunktion in diesem Block nicht imperativ, das heißt als eine Folge von nacheinander auszuführenden Befehlen beziehungsweise Rechenschritten, sondern als eine Menge von Grundoperationen und den Datenfluss zwischen diesen, so dass diese Definition möglichst analog zu der mathematischen Beschreibung der Schrittfunktion der Klasse von ODE-Verfahren erfolgen kann. Deswegen darf in dem Stepfunction-Block ein Vektor auch nur einmal als das Ziel einer Zuweisung dienen, wodurch ein Vektor auch nur das Ergebnis einer einzigen Vektorgrundoperation abspeichern kann. Dafür kann ein Vektor aber auch bereits an einer Stelle im Stepfunction-Block als das Argument einer Vektorgrundoperation dienen, bevor diesem Vektor im Stepfunction-Block das Ergebnis einer Vektorgrundoperation zugewiesen wurde. Als Vektorgrundoperation selbst stehen (RHS | LC | MAP | RED)-Operationen zur Verfügung. Falls die Anzahl an Vektorgrundoperationen von den zeitinvarianten Konstanten eines spezifischen ODE-Verfahrens abhängig sein soll, muss in der Klassen-DSL für diese Vektorgrundoperationen die Zuweisung der Argumente über For-Schleifen realisiert werden. Analog, falls die Anzahl an Argumenten einer LC-Operation von den zeitinvarianten Konstanten eines spezifischen ODE-Verfahrens abhängig sein soll, muss dies in der Klassen-DSL über eine For-Schleife innerhalb der LC-Operation realisiert werden. Um in der Klassen-DSL zu beschreiben, dass eine Vektorgrundoperation einen Vektor eines vergangenen Zeitschritts als Argument referenziert, muss diese Referenz mit einer Zeitschrittdistanz versehen werden. Abschließend gilt wieder anzumerken, dass bei expliziten ODE-Verfahren in dieser so definierten Schrittfunktion keine zyklischen Abhängigkeiten zwischen den Vektorgrundoperationen auftreten, wodurch die Schrittfunktion bei einem expliziten ODE-Verfahren wiederum nach der topologischen Sortierung der Grundoperationen direkt als Berechnungsvorschrift dienen kann. Dahingegen treten bei impliziten ODE-Verfahren in der Schrittfunktion solche zyklischen Abhängigkeiten auf, wodurch die Schrittfunktion die Lösung eines nichtlinearen Gleichungssystem erfordert.

11.11.2 Verfahren-DSL

Die *Verfahren-DSL* (siehe Abbildung 11.11) dient zur Definition eines ODE-Verfahrens einer zuvor definierten Klasse von ODE-Verfahren. Dementsprechend besteht ein Dokument dieser DSL nur aus einem Meta-Block, welcher die Meta-Informationen des spezifischen ODE-Verfahrens beinhaltet, und einem Constants-Block, welcher die zeitinvarianten Konstanten des spezifischen ODE-Verfahrens beinhaltet. Für die Initialisierung der Arrays von zeitinvarianten Konstanten verwendet die Verfahren-DSL sogenannte Initializer-Lists.

11.11.3 Implementierung der DSL

Als Proof-Of-Concept entwickelten wir für die Erzeugung eines Datenflussgraphen aus je einem Dokument der Klassen-DSL und der Verfahren-DSL ein Python-Programm. Als Parser-Framework für unser Python-Programm entschieden wir uns für die Bibliothek *LARK* [W21]. Dementsprechend entwickelten wir für unsere DSL eine *LARK-Grammatik*. Für die Ausgabe des Programms entschieden wir uns für eine Datei im DOT-Format [W22], welches ein standardisiertes, menschenlesbares, leicht zu parsendes Dateiformat für Graphen ist. Dementsprechend führt das Python-Programm insgesamt folgende Schritte aus:

1. Parsen der Dokumente der DSL mit der LARK-Bibliothek.
2. Transformation des abstrakten Syntaxbaums in einen Datenflussgraphen.
3. Abspeichern des Datenflussgraphen als DOT-Datei.

```
classname = "Explicit RK-methods"

meta:
    string name
    int order
    int stages (s)

constants:
      double a[s][s]
      double b[s]
      double c[s]

vectors:
      systemstate y
      systemstate Y[s]
      derivative  F[s]

stepfunction:
   for (i in 1 ... s)
   {
      Y[s] = LC(y<1>, for (j in 1 ... i):  h_k  * a[i][j] * F[i])
      F[s] = RHS(Y[j], t_k + c[i] * h_k)
   }
   y = LC(y<1>, for (i in 1 ... s):  h_k * b[i] * F[i])
```

Abbildung 11.10: Codebeispiel für die Klassen-DSL. Diese Abbildung beschreibt die Klasse der expliziten RK-Verfahren mit der selbst entwickelten Klassen-DSL. Dabei wird im Stepfunction-Block die Zeitschrittdistanz bei der Referenzierung eines Vektors über die spitzen Klammern <...> realisiert.

```
meta:
      name = "RK3/8"
      stages = 4

constants:
    a =
      {
         {                     }
         {  1/3,               }
         { -1/3,    1,         }
         {    1,   -1,    1,   }
      }

      b =
         {  1/8, 3/8, 3/8, 1/8}

      c =
         {    0, 1/3, 2/3,   1}
```

Abbildung 11.11: Codebeispiel für die Verfahren-DSL. Diese Abbildung zeigt die Definition der Konstanten von dem RK-3/8-Verfahren, ein explizites RK-Verfahren mit 4 Stufen.

Im Rahmen dieser Arbeit implementierten wir noch keine Schnittstelle zwischen dem Modul der DSL und dem Modul des Datenflussgraphen, durch welche das Modul des Datenflussgraphen einen mit der DSL beschriebenen Datenflussgraphen aus einer Datei laden und weiter verarbeiten kann. Der dazu erforderliche Aufwand ist jedoch rein technischer Natur.

12 Schlussteil

12.1 Zusammenfassendes Fazit

12.1.1 Fazit zur Datenflussgraphrepräsentation

In dieser Arbeit zeigten wir zunächst, dass sich der Zeitschritt von vielen häufig verwendeten ODE-Verfahren mit einer Datenflussgraphrepräsentation abstrahieren lässt. Ein Datenflussgraph dieser Repräsentation besteht dabei im Wesentlichen aus (RHS | LC | MAP | RED)-Operationen als Vektorgrundoperationen. Um in einem solchen Datenflussgraphen zu definieren, ob sich eine Abhängigkeit auf den aktuellen oder den vorherigen Zeitschritt bezieht, annotieren wir die Abhängigkeiten zwischen den Vektorgrundoperationen im Datenflussgraphen mit Zeitschrittdistanzen. Ebenso zeigten wir, dass der Datenflussgraph für viele häufig verwendete explizite ODE-Verfahren, wie zum Beispiel RK-Verfahren, PIRK-Verfahren oder Adams-Bashforth-Verfahren, im Wesentlichen aus einer RHS $\rightarrow$ LC-Abhängigkeitskette besteht, welche sich wiederum aus RHS $\rightarrow$ LC-Gliedern zusammensetzt.

12.1.2 Fazit zur Basisimplementierung

Über diese Datenflussgraphrepräsentation eines ODE-Verfahrens konnten wir leicht eine Basisimplementierung für GPUs und CPUs erzeugen. Diese Basisimplementierung startet ein Kernel für jede Vektorgrundoperation im Datenflussgraphen und nutzt innerhalb eines jeden dieser Kernels die Systemparallelität der Vektorgrundoperation aus, um die Berechnungen auf die Rechenkerne des jeweiligen Prozessors zu verteilen.

Die von uns durchgeführten Experimente zeigen, dass für dichtbesetzte Anfangswertprobleme die RHS-Funktion die Laufzeit eines jeden ODE-Verfahrens dominiert, und sie zudem, falls sie auch eine hohe arithmetische Intensität besitzt, durch die Rechenleistung des Prozessors gebunden ist. So lastet zum Beispiel das getestete RHS-Kernel des NBODY-Problems auf der getesteten Volta-GPU die DP-Rechenwerke zu 81.7 % und die DRAM-Bandbreite nur zu 0.3 % aus, während es auf der getesteten Skylake-CPU die DP-Rechenleistung nur zu 12.9 % ausnutzt und die DRAM-Bandbreite zu 1.9 % auslastet. Auch zeigen die Experimente, dass für dünnbesetzte Anfangswertprobleme mit einer geringen arithmetischen Intensität, wie dem BRUSS2D-Problem, die Basisimplementierung eines ODE-Verfahrens stark durch die DRAM-Bandbreite gebunden ist. So lastet zum Beispiel die Basisimplementierung vom Verner-Verfahren mit BRUSS2D als Testproblem auf der getesteten Maxwell-GPU die IPC nur zu 5.0 % und die DRAM-Bandbreite zu 86 % aus, während es auf der getesteten Skylake-CPU die DP-Rechenleistung nur zu 0.8 % und die DRAM-Bandbreite zu 75 % auslastet. Dies ist darauf zurückzuführen, dass bereits bei mittelgroßen Problemgrößen die Caches des Prozessors zu klein sind, um Daten zwischen zwei aufeinanderfolgenden Kernels wiederverwenden zu können. Somit sind für dünnbesetzte Anfangswertprobleme mit einer geringen arithmetischen Intensität zusätzliche Lokalitätsoptimierungen nötig, um eine gute Performance zu erzielen.

12.1.3 Fazit zur Kernelfusion

Als Nächstes zeigten wir, dass sich mit dem Ansatz der Kernelfusion automatisch Implementierungsvarianten aus der Datenflussgraphrepräsentation eines ODE-Verfahrens für GPUs und CPUs generieren lassen. Diese per Kernelfusion generierten Implementierungsvarianten zielen

auf eine im Vergleich zu einer Basisimplementierung erhöhte Datenwiederverwendung ab, um darüber für dünnbesetzte Anfangswertprobleme mit einer geringen arithmetischen Intensität die Performance deutlich zu erhöhen. Hierfür fusionierten wir im Wesentlichen sämtliche Vektorgrundoperationen in einem RHS $\rightarrow$ LC-Glied zu einem Kernel. Zusätzlich stellten wir drei Enabling-Transformationen vor, mit Hilfe derer sich die Datenwiederverwendung und dadurch die Performance weiter verbessern ließen. Danach gingen wir darauf ein, wie sich sowohl auf GPUs als auch auf CPUs aus einem fusionierten Kernel ein optimierter Code erzeugen lässt, welcher den On-Chip-Speicher und sonstige Besonderheiten der jeweiligen Architektur effizient ausnutzt. Deshalb setzt der Code eines fusionierten GPU-Kernels primär auf die Datenwiederverwendung über den Registersatz, während der Code eines fusionierten CPU-Kernels primär auf die Datenwiederverwendung über die Caches setzt.

Die anschließend durchgeführten Experimente mit den per Kernelfusion und Enabling-Transformationen generierten Varianten mit BRUSS2D als Testproblem zeigten sowohl auf den untersuchten GPUs und CPUs ein sehr ähnliches Gesamtbild. Auf den getesteten GPUs lässt sich durch die Kernelfusion und die Enabling-Transformationen eine Beschleunigung zwischen 1.62 und 3.32 gegenüber der Basisimplementierung erzielen, während sich auf den getesteten CPUs eine Beschleunigung zwischen 1.98 und 4.14 gegenüber der Basisimplementierung erzielen lässt. Diese Beschleunigung ist darauf zurückzuführen, dass die Kernelfusion und die Enabling-Transformationen das Datenvolumen beziehungsweise das DRAM-Volumen je nach ODE-Verfahren auf 50 % bis 25 % gegenüber der jeweiligen Basisimplementierung reduzieren können. Allerdings zeigten die Untersuchungen auch, dass die per Kernelfusion und Enabling-Transformationen generierten Implementierungsvarianten für dünnbesetzte Anfangswertprobleme mit einer niedrigen arithmetischen Intensität weiterhin meist stark durch die DRAM-Bandbreite gebunden sind. Dadurch lasten sie die Rechenleistung des jeweiligen Prozessors nur schlecht aus. So beträgt zum Beispiel auf der Maxwell-GPU mit BRUSS2D als Testproblem die Auslastung der IPC je nach Variante und Verfahren zwischen 5 % und 15 % sowie die Auslastung der DRAM-Bandbreite zwischen 75 % und 87 %. Ebenso beträgt zum Beispiel auf der Skylake-CPU mit BRUSS2D als Testproblem die Ausnutzung der DP-Peak-Performance je nach Variante und Verfahren zwischen 1.2 % und 3.5 % sowie die Auslastung der DRAM-Bandbreite zwischen 73 % und 82 %. Somit sind zusätzliche Lokalitätsoptimierungen nötig, um die Performance eines ODE-Verfahrens gegenüber einer per Kernelfusion generierten Implementierungsvariante weiter zu erhöhen.

12.1.4 Fazit zum Tiling

Da das beliebige Zugriffsmuster einer RHS-Operation bei einem allgemeinen Anfangswertproblem die Fusion einer LC $\rightarrow$ RHS-Abhängigkeit und damit die Datenwiederverwendung entlang einer solchen LC $\rightarrow$ RHS-Abhängigkeit verhindert, spezialisierten wir uns als Nächstes auf Anfangswertprobleme mit einer beschränkten Zugriffsdistanz. Selbst mit dieser beschränkten Zugriffsdistanz sind jedoch LC $\rightarrow$ RHS-Abhängigkeiten nicht direkt zu einem einzigen Kernel, sondern nur über eine partitionierte Fusion zu einem Kernelpaar oder zu einem Kerneltriplet verschmelzbar. Diese partitionierte Fusion führte nun zu zwei Tiling-Schemata, und zwar zu dem hexagonalen Tiling-Schema und dem trapezoidalen Tiling-Schema. Wir stellten nicht nur vor, wie wir diese Tiling-Schemata über die Stufen eines ODE-Verfahrens, sondern auch über dessen Zeitschritte, verlaufen lassen können. Danach beschrieben wir nicht nur das Single-Core-Tiling, bei welchem jeder Prozessorkern an seinem eigenen Tile arbeitet, sondern auch das Multi-Core-Tiling, bei welchen mehrere Kerne an einem Tile zusammenarbeiten. Durch

dieses Multi-Core-Tiling finden wiederum größere Tile-Größen in den On-Chip-Speicher des Prozessors Platz, wodurch das Multi-Core-Tiling für größere Zugriffsdistanzen deutlich effizienter als das Single-Core-Tiling wird. Ebenso gingen wir darauf ein, wie man für GPUs und CPUs effizienten Code für Kernels mit Tiling erzeugen kann, so dass das Kernel effizient die On-Chip-Speicher der jeweiligen Architektur ausnutzt. Hierbei zeigte sich wiederum, dass die GPU-Kernels mit Tiling sich auf die Datenwiederverwendung über den Registersatz der GPU fokussieren müssen, während die CPU-Kernels sich auf die Datenwiederverwendung über die Caches der CPU fokussieren müssen.

Letztlich führten wir experimentelle Untersuchungen auf GPUs und CPUs für Implementierungsvarianten mit Tiling durch. Diese Untersuchungen zeigen, dass sich mit dem Tiling gegenüber der Kernelfusion das DRAM-Volumen deutlich reduzieren und die Performance deutlich steigern lässt. So erzielten für die getesteten ODE-Verfahren die Implementierungsvarianten mit Tiling gegenüber einer Referenzimplementierung nur mit Fusion bei kleinen Zugriffsdistanzen auf den getesteten GPUs eine Beschleunigung von 1.8 bis 8.2 und auf den getesteten CPUs eine Beschleunigung von 7.5 bis 14.1. Ebenso zeigten die Untersuchungen, dass bereits bei kleinen bis mittelgroßen Zugriffsdistanzen das Multi-Core-Tiling deutlich performanter als das Single-Core-Tiling ist. So erzielt zum Beispiel auf der getesteten Volta-GPU beim Verner-Verfahren das Single-Workgroup-Tiling nur bis zu einer beschränkten Zugriffsdistanz von 2 048 eine Beschleunigung gegenüber der Referenzimplementierung, während das Multi-Workgroup-Tiling bei einer beschränkten Zugriffsdistanz von 131 072 noch eine Beschleunigung von 1.4 gegenüber der Referenzimplementierung erzielt. Ebenso erzielt zum Beispiel auf der getesteten Skylake-CPU beim Verner-Verfahren das Single-Thread-Tiling nur bis zu einer beschränkten Zugriffsdistanz von 32 768 eine Beschleunigung gegenüber der Referenzimplementierung, während das Multi-Thread-Tiling noch bei einer beschränkten Zugriffsdistanz von 131 072 noch eine Beschleunigung von 1.6 gegenüber der Referenzimplementierung erzielt.

12.1.5 Fazit zum Framework

Zuletzt stellten wir vor, wie wir unseren Ansatz zum Lösen von Anfangswertproblemen in einem Framework umsetzten. Dieses Framework generiert zunächst den Datenflussgraphen von einem durch einen Benutzer gegebenen ODE-Verfahren einer unterstützen Klasse. Als Nächstes generiert es je nach Benutzervorgabe aus dem Datenflussgraphen die Basisimplementierung und deren Kernels, eine Implementierungsvariante mit Kernelfusion oder eine Implementierungsvariante mit Tiling und führt diese aus. Ebenso beherrscht dieses Framework eine Vielzahl von weiteren nützlichen Funktionalitäten, wie automatisches Speichermanagement, NUMA-Unterstützung sowie eine Autotuning-Funktionalität für das Tiling. Auch ist es einem Benutzer des Frameworks leicht möglich, neue Klassen von ODE-Verfahren entweder durch Erweiterung des C++-Codes des Frameworks oder über eine DSL innerhalb des Frameworks zum Framework hinzuzufügen. Dementsprechend demonstriert dieses Framework noch einmal, dass sich der Ansatz der Datenflussgraphrepräsentation von ODE-Verfahren dazu eignet, elegant und automatisch Implementierungsvarianten mit Fusion und Tiling für ein gegebenes ODE-Verfahren und eine gegebene Zielarchitektur zu erzeugen, welche auf das jeweilige ODE-Verfahren und auf die jeweilige Zielarchitektur optimiert sind. Ebenso demonstriert das Framework die leichte Erweiterbarkeit dieses Ansatzes der Datenflussgraphrepräsentation auf neue ODE-Verfahren und Architekturen.

12.1.6 Abschließendes Fazit

Somit können wir als Fazit ziehen, dass sich der Ansatz der Datenflussgraphrepräsentation für ODE-Verfahren eignet, um für ODE-Verfahren automatisch effiziente Implementierungsvarianten für GPUs und CPUs zu generieren. Am einfachsten lässt sich über die Datenflussgraphrepräsentation eine Basisimplementierung generieren, welche die Vektorgrundoperationen nacheinander ausführt. Ebenso lässt sich die Technik der Kernelfusion auf die Datenflussgraphrepräsentation anwenden, um Implementierungsvarianten zu generieren, welche auf Grund ihrer erhöhten Datenwiederverwendung gegenüber der Basisimplementierung auf CPUs und GPUs eine große Beschleunigung für dünnbesetzte Anfangswertprobleme mit einer niedrigen arithmetischen Intensität erzielen. Eine zusätzliche Spezialisierung auf Anfangswertprobleme mit beschränkter Zugriffsdistanz ermöglicht es, aus der Datenflussgraphrepräsentation eines ODE-Verfahrens Implementierungsvarianten mit Tiling zu generieren, welche wiederum auf CPUs und GPUs wegen ihrer erhöhten Datenwiederverwendung gegenüber den nur per Kernelfusion generierten Varianten eine deutlich verbesserte Performance besitzen.

Ein wichtiges Fazit ist zudem, dass man bei der Generierung von Implementierungsvarianten für GPUs und CPUs die Besonderheiten der jeweiligen Architektur beachten muss. Berücksichtigt man jedoch diese Besonderheiten, so zeigte sich in dieser Arbeit, dass zumindest bei den von uns generierten Varianten von ODE-Verfahren, das heißt bei den Varianten der Basisimplementierung und bei den Varianten mit Kernelfusion und Tiling, sich der Code auf der jeweiligen Architektur dann in etwa so verhält, wie es die Eckdaten der Architektur, also im Wesentlichen die DP-Performance, die DRAM-Bandbreite, die Kernzahl und die On-Chip-Speichergröße, erwarten lassen.

12.2 Erfüllung der Ziele

Zusammenfassend können wir sagen, dass der von uns entwickelte Ansatz über eine Datenflussgraphrepräsentation von einem ODE-Verfahren optimierte Implementierungsvarianten mit Kernelfusion oder Tiling zu generieren, die Ziele wie folgt abdeckt:

- **Unterstützung eines breiten Spektrums an häufig verwendeter expliziter ODE-Verfahren:** Über unsere Datenflussgraphrepräsentation von ODE-Verfahren und die automatische Generierung von Implementierungsvarianten daraus unterstützen wir nicht nur das Lösen von Anfangswertprobleme durch RK-Verfahren, PIRK-Verfahren, Peer-Verfahren oder Adams-Bashforth-Verfahren, sondern das generelle Lösen von Anfangswertproblemen über die Klasse der allgemeinen linearen Verfahren.
- **Effizientes Lösen von Anfangswertproblemen ohne beschränkte Zugriffsdistanz:** Dichtbesetzte Anfangswertprobleme und dünnbesetzte Anfangswertprobleme mit einer hohen arithmetischen Intensität lassen sich bereits mit der Basisimplementierung effizient lösen. Dahingegen lassen sich dünnbesetzte Anfangswertprobleme mit einer geringen arithmetischen Intensität zwar auch über die Basisimplementierung lösen, jedoch ist für solche Anfangswertprobleme eine per Kernelfusion generierte Implementierungsvariante auf Grund des höheren Grades an Datenwiederverwendung und der daraus resultierenden größeren Performance besser geeignet.
- **Effizientes Lösen von Anfangswertproblemen mit beschränkter Zugriffsdistanz:** Hierfür stehen nicht nur die Basisimplementierung zur Verfügung, sondern auch die Va-

rianten mit Kernelfusion sowie mit Tiling, wobei eine Variante mit Tiling für kleinere, mittlere und große Zugriffsdistanzen eine bessere Performance erzielt als eine Variante mit Kernelfusion oder die Basisimplementierung. Lediglich bei sehr großen Zugriffsdistanzen ist eine Variante mit Kernelfusion besser geeignet als eine Variante mit Tiling.

- **Ausnutzung von Systemparallelität und Methodenparallelität:** Alle unserer Varianten nutzen die Systemparallelität effizient aus, um die Berechnungen sowohl auf GPUs als auch auf CPUs auf die Rechenkerne zu verteilen. Zusätzlich nutzen sämtliche Varianten sowohl auf GPUs als auch auf CPUs die Systemparallelität für ihre Vektorisierung aus. Die Methodenparallelität nutzen unsere Varianten sowohl bei der Kernelfusion sowie beim Tiling aus, indem sie die unabhängigen Stufen eines ODE-Verfahrens fusionieren beziehungsweise in dasselbe RHS $\rightarrow$ LC-Glied der Abhängigkeitskette des ODE-Verfahrens einfügen, um dadurch die Datenwiederverwendung zu erhöhen. Diese Ausnutzung der Methodenparallelität für die Datenwiederverwendung ist zwar für viele ODE-Verfahren sehr effektiv, entfremdet jedoch die Methodenparallelität ihrem ursprünglichen Verwendungszweck, und zwar eine zusätzliche Quelle an Parallelität zu bieten, um mehr Rechenkerne für das Lösen eines Anfangswertproblems benutzen zu können.
- **Ein hoher Grad an Datenwiederverwendung über die On-Chip-Speicher:** Wir erzielten sowohl über die Kernelfusion als auch über das Tiling, dass ein Prozessor in einem Kernel möglichst viele Daten über seinen On-Chip-Speicher wiederverwenden kann und möglichst wenig Daten zwischen den Kernels über den DRAM transferieren muss. Dabei gehen sowohl unser Ansatz mit der Kernelfusion als auch unser Ansatz mit dem Tiling auf die architekturspezifischen Besonderheiten der On-Chip-Speicher von GPUs und CPUs ein. Diese durch die Kernelfusion und das Tiling erhöhte Datenwiederverwendung zeigt sich in einem deutlich reduzierten Datenvolumen und DRAM-Volumen der per Kernelfusion und Tiling generierten Varianten.
- **Unterstützung von GPUs und CPUs als optimierte Zielplattformen:** Wir können über unserem Ansatz beziehungsweise über unser Framework für CPUs und GPUs sowie für ein gegebenes ODE-Verfahren nicht nur die Basisimplementierung generieren, sondern auch die Varianten mit Fusion und mit Tiling. Dabei erzeugt unser Ansatz bei der Generierung einer Variante einen Maschinencode, welcher auf die jeweilige Architektur optimiert ist.

12.3 Ausblick

Aus den erzielten Ergebnissen dieser Arbeit ergeben sich folgende weitere interessante Ansätze, deren wissenschaftliche Untersuchung lohnenswert erscheint:

- **Ausbau des Autotunings für die Kernelfusion:** Aktuell muss ein Benutzer des Frameworks zur Erzeugung von Implementierungsvarianten die Enabling-Transformationen von Hand auf ein gegebenes ODE-Verfahren anwenden. Dies ist nicht nur umständlich, sondern es fällt einem Benutzer gerade bei einem ODE-Verfahren mit vielen Stufen schwer, die Enabling-Transformationen so anzuwenden, dass sie das Datenvolumen minimieren. Deshalb wäre hier ein Automatismus sinnvoll, welcher die Enabling-Transformationen automatisch auf den Datenflussgraphen eines beliebigen

ODE-Verfahrens anwendet, so dass die Enabling-Transformationen das Datenvolumen minimieren.

- **Ausbau des Autotunings für das Tiling:** Man könnte unseren Ansatz des Autotunings für das Tiling noch weiter verfeinern, so dass er noch weniger Kernels einem Benchmark unterziehen muss. Dieses Verfeinern könnten wir zum Beispiel durch ein Ausästen des Suchraums, zum Beispiel über Performance-Modelle, realisieren. Ebenso könnten wir dieses Autotuning noch mit der Splitting-Transformation und der Cloning-Transformation, welche wir beim Tiling nicht anwendeten, kombinieren. In unserer Arbeit realisierten wir das Autotuning zudem nur über einen Offline-Ansatz, wodurch die während des Autotunings ausgeführten Kernels nicht zum Lösen des Anfangswertproblems beitragen. Dementsprechend ist eine Erweiterung des Autotunings um einen Online-Ansatz, bei welchem die vom Autotuning ausgeführten Kernels zum Lösen des Anfangswertproblems beitragen könnten, vielversprechend.

- **Erweiterung unseres Frameworks auf heterogene Cluster:** Um nicht nur die Rechenleistung einer CPU oder GPU, sondern eines gesamten Clusters auszunutzen, ist eine Erweiterung unseres Frameworks für heterogene Cluster sowohl für dichtbesetzte Anfangswertprobleme ohne beschränkte Zugriffsdistanz als auch für dünnbesetzte Anfangswertprobleme mit beschränkter Zugriffsdistanz vielversprechend:

  - **Cluster-Implementierung für dichtbesetzte Anfangswertprobleme:** Für Anfangswertprobleme ohne beschränkte Zugriffsdistanz müsste man für eine effiziente Cluster-Implementierung die Systemkomponenten blockweise auf die Prozessoren des Clusters aufteilen, und vor jeder RHS-Operation den Argumentvektor über eine All-Gather-Operation zwischen allen Prozessoren synchronisieren. Da bei dieser blockweisen Aufteilung für dichtbesetzte Anfangswertprobleme die RHS-Operation gegenüber der All-Gather-Operation die Laufzeit dominieren würde, würde sich mit diesem Ansatz bei solchen dichtbesetzten Anfangswertproblemen eine gute Performance erzielen lassen. Bei dünnbesetzten Anfangswertproblemen jedoch würde bei einer solchen blockweisen Aufteilung die All-Gather-Operation vor jeder RHS-Operation gegenüber allen Vektorgrundoperationen die Laufzeit dominieren, wodurch sich die Rechenleistung des Clusters wiederum nur schlecht ausnutzen ließe. Allerdings verhindert das beliebige Zugriffsmuster der RHS-Operation bei einem allgemeinen Anfangswertproblem eine Reduzierung des Kommunikationsaufwands. Deshalb ist auf einem Cluster keine effiziente Implementierung eines Lösers für dünnbesetzte Anfangswertprobleme möglich, ohne das Zugriffsmuster der RHS-Operation, wie zum Beispiel über eine beschränkte Zugriffsdistanz, einzuschränken.
  - **Cluster-Implementierung für dünnbesetzte Anfangswertprobleme mit beschränkter Zugriffsdistanz:** Bei dünnbesetzten Anfangswertproblemen mit beschränkter Zugriffsdistanz müsste man für eine effiziente Cluster-Implementierung wieder die Systemkomponenten blockweise auf die Prozessoren des Clusters aufteilen. Allerdings würde man vor jeder RHS-Operation keine All-Gather-Operation benötigen, sondern jeder Prozessor müsste nur diejenigen Systemkomponenten, die sich innerhalb der Zugriffsdistanz am Rand seines Blockes befinden, mit den Nachbarprozessoren synchronisieren. Dadurch würden in dieser Cluster-Implementierung nur noch benachbarte Prozessoren miteinander kommunizieren.

Zusätzlich wäre es hier bei einem heterogenen Cluster möglich, dass ein jeder Prozessor die Systemkomponenten in seinem Block über seine eigne Implementierungsvariante berechnet, welche für seine Architektur am effizientesten ist. So wäre es zum Beispiel möglich, dass eine GPU ihren Block über eine per Kernelfusion generierte Variante berechnet, während eine benachbarte CPU ihren Block über eine Variante mit einem hexagonalen oder trapezoidalen Tiling berechnet. Ein Prozessor könnte am Rand seines Blockes die Systemkomponenten mit den benachbarten Prozessoren synchronisieren, während er die Systemkomponenten in der Mitte seines Blockes berechnet. Hierdurch könnte eine Cluster-Implementierung sowohl die Netzwerkbandbreite als auch die Rechenleistung des Clusters gleichzeitig ausnutzen.

Potentiell müsste eine Cluster-Implementierung gerade bei heterogenen Clustern noch eine Lastbalancierung implementieren, das heißt, sie muss für jeden Prozessors je nach Last des Prozessors und der Last der beiden benachbarten Prozessoren den Block des Prozessors vergrößern oder verkleinern.

- **Numerische Untersuchungen:** In dieser Arbeit führten wir nur Untersuchungen durch, wie sehr mehrere gegebene ODE-Verfahren auf einer Vielzahl an Test-GPUs und Test-CPUs durch unsere Optimierungen beschleunigt werden. Allerdings wäre es interessant zu untersuchen, welches ODE-Verfahren auf modernen GPUs und CPUs unter Einhaltung eines gegebenen lokalen Diskretisierungsfehlers am schnellsten oder am energieeffizientesten ist. Auch wäre hier eventuell ein Autotuning-Ansatz lohnenswert, welcher in kurzer Zeit eventuell mit Hilfe von Performance- oder Energiemodellen das effizienteste ODE-Verfahren mit seiner effizientesten Implementierungsvariante bestimmt.
- **Weitere Spezialisierungen für weitere Arten von Anfangswertproblemen:** Aktuell beherrscht unser Framework nur allgemeine Anfangswertprobleme und besitzt nur eine Spezialisierung auf Anfangswertprobleme mit einem 2D- oder 3D-Iterationsraum für die Kernelfusion sowie eine Spezialisierung auf Anfangswertprobleme mit beschränkter Zugriffsdistanz für das Tiling. Allerdings gibt es potentiell noch viele weitere Arten beziehungsweise Eigenschaften von Anfangswertproblemen, auf die sich unser Framework spezialisieren könnte. Insbesondere scheint hier zunächst eine Spezialisierung auf 2D- oder 3D-Stencils vielversprechend zu sein, da diese für viele Anwendungen sehr bedeutsam sind. Für solche 2D- oder 3D-Stencils könnte unser Framework mit der zusätzlichen Zeitdimension von ODE-Verfahren über ein 3D- beziehungsweise 4D-Tiling die Speicherzugriffslokalität und damit die Performance deutlich verbessern. Ebenso scheint eine Spezialisierung auf Partikelsimulationen lohnenswert, da sich bei diesen mit der Hilfe von zusätzlichen Datenstrukturen die Rechenkomplexität in Abhängigkeit von der Systemgröße n von $O(n^2)$ auf $O(n \log(n))$ reduzieren ließe.
- **Integration der DSL in unser Framework und deren Erweiterung:** Selbst wenn wir bereits eine einfache DSL für die Beschreibung eines ODE-Verfahrens entwickelten, um daraus den Datenflussgraphen des ODE-Verfahrens zu generieren, so ist diese DSL noch nicht vollständig in unser Framework integriert. Jedoch würde eine solche Integration es einem Benutzer deutlich erleichtern, selbst neue ODE-Verfahren zu unserem Framework hinzuzufügen. Des Weiteren wäre eine Erweiterung dieser DSL sinnvoll, so dass ein Benutzer mit dieser die zu generierende Implementierungsvariante näher spezifizieren kann, das heißt, ob unser Framework die Basisimplementierung oder eine Variante

mit Kernelfusion oder mit Tiling generieren soll. Auch scheint eine Erweiterung der DSL um eine Sprache zur Spezifikation von Anfangswertproblemen vielversprechend, da hierdurch ein Benutzer ebenfalls leicht neue Anfangswertprobleme zu unserem Framework hinzufügen könnte. Allerdings wäre bei dieser Art der Erweiterung der DSL darauf zu achten, dass die DSL nicht zu einer GPL wird.

13 Anhang

13.1 GPU-APIs mit CPU-Unterstützung

Mehrere APIs zur GPU-Programmierung, wie zum Beispiel OpenCL [W10], C++ AMP [T21] oder hip [W14], beziehungsweise deren Implementierungen bieten einem Programmierer auch die Möglichkeit, die Berechnungen beziehungsweise die Kernels nicht nur von einer GPU, sondern auch von einer Mehrkern-CPU ausführen zu lassen. Nichtsdestotrotz liegt der Fokus dieser APIs weiterhin auf GPUs. Deshalb übernehmen diese GPU-APIs für Ihre CPU-Unterstützung das Konzept des Kernels als ein paralleles Programm, welches nicht nur von GPUs, sondern auch von CPUs ausgeführt werden kann. Dabei besteht ein solches CPU-Kernel wiederum aus vielen potentiell parallelen Instanzen der Kernelfunktion, die weiterhin als Workitems bezeichnet werden und die weiterhin zu Workgroups gruppiert sind.

Startet eine Anwendung über eine GPU-API ein Kernel auf einer Mehrkern-CPU, so lässt die GPU-API von der Mehrkehr-CPU des Kernel wie folgt abarbeiten (siehe zum Beispiel [T27], [T25] und [T15]): Zuerst initialisiert die GPU-API einen Task-Pool bestehend aus den Workgroups des Kernels und startet für jeden logischen CPU-Kern einen Worker-Thread. Nach ihrem Starten entfernen die Worker-Threads die Workgroups aus dem Task-Pool und führen sie aus, so dass ein jeder Worker-Thread jeweils nur eine Workgroup gleichzeitig ausführt. Für diese Ausführung iteriert der Worker-Thread über die Workitems der Workgroup in einer Schleifenstruktur, die nicht von der GPU-API spezifiziert wird. Dadurch kann der CPU-Compiler der GPU-API die Schleifenstruktur innerhalb einer Workgroup auf das entsprechende Kernel und auf die Ziel-CPU-Architektur hin optimieren. Dies hat zur Folge, dass ein Programmierer nur kaum die generierten Schleifenstrukturen kontrollieren kann. Der Compiler kann zudem Operationen innerhalb eines einzigen Workitems vektorisieren, aber auch Operationen aus unterschiedlichen Workitems innerhalb derselben Workgroup. Zusätzlich kann der Compiler weitere Schleifenoptimerungen durchführen, wie etwa das Abrollen des Iterierens einer Workgroup über ihre Workitems. Während ein CPU-Compiler keine Parallelität auf Thread-Ebene zwischen den Workitems einer Workgroup einführt, so nutzt ein CPU-Compiler dennoch die entspannte Speicherkonsistenz, welche aus der Nebenläufigkeit der Workitems einer Workgroup resultiert, für Optimierungen aus. Einem Programmierer ist es bei GPU-APIs mit CPU-Unterstützung zudem mit Hilfe eines Compiler-Hinweises möglich, die Anzahl an Workitems pro Workgroup auf eins zu beschränken. Durch dieses Beschränken wird die Kernelfunktion sehr ähnlich zu einer Eintrittsfunktion für einen CPU-Thread in einer General-Purpose-Programmiersprache. Aber im Gegenzug verliert der Programmierer denjenigen Vorteil, dass der CPU-Compiler das Kernel nun nicht mehr effizient über die Workitems einer Workgroup hinweg vektorisieren und optimieren kann.

13.2 Alternative Möglichkeiten zur Integration der CPU-Unterstützung in unser Framework

Statt dem DLL-Ansatz hätten wir CPUs als Zielplattform für unser Framework auch per OpenCL (siehe Abschnitt 13.1 wie OpenCL von CPUs implementiert wird) oder einem Kernel-Interpreter unterstützen können, wobei die beiden alternativen Ansätze in unseren Augen kaum Vorteile, dafür viele Nachteile besitzen:

CPU-Unterstützung über OpenCL: Während wir in unserem Framework den OpenCL-Pfad

des Moduls der Ausführung, den wir für GPUs implementierten, auch für CPUs hätten verwenden können, so hätten wir für die effiziente CPU-Unterstützung per OpenCL ebenfalls einen neuen Codegenerator entwickeln müssen, der für CPUs optimierten Quelltext in OpenCL-C erzeugt. Bei dieser Codegenerierung in OpenCL-C ist aber Folgendes problematisch:

- Der Ansatz den Iterationsraum in einem Block beziehungsweise in einem Tile über mehrere Workitems per Workgroup zu definieren wäre keine Option gewesen, da wir hierdurch die Kontrolle über die Schleifenstruktur in einem solchen Block beziehungsweise Tile verloren hätten. Ebenso hätte hierdurch ein CPU-Kernel eine RHS-Operation nur über die Eval-Comp-Implementierung und nicht über die Eval-Range-Implementierung der RHS-Funktion berechnen können.
- Während der Ansatz von nur einem Workitem per Workgroup für CPUs passender wäre, so würden wir hierbei auch fast alle Vorteile des Programmiermodells von OpenCL-C verlieren. Da zudem der Ansatz mit einem Workitem per Workgroup nicht der primäre beabsichtigte Use-Case von OpenCL auf CPUs ist, ist es zudem möglich, dass dieser Ansatz von einem OpenCL-CPU-Compiler nicht effizient umgesetzt wird.
- Die Programmiersprache C wurde im Jahr 1972 entwickelt, während OpenCL beziehungsweise OpenCL-C erst im Jahre 2008 entwickelt wurden. Zudem besitzt C einen deutlich größeren Marktanteil als OpenCL. So existieren zum Beispiel auf GitHub [W23] im Jahre 2022 circa 1 000 000 Repositories mit der Programmiersprache C, aber nur 19 000 Repositories, die OpenCL verwenden. Auch deckt C als General-Purpose-Programming-Language sämtliche Anwendungen ab, während OpenCL auf die Programmierung heterogener Systeme spezialisiert ist (siehe [T25]). Folglich kann man davon ausgehen, dass gewöhnliche C-Compiler bis in ferner Zukunft häufig aktualisiert und für neue CPUs optimiert werden. Im Gegensatz dazu ist es ungewiss, wie lange OpenCL-Implementierungen für CPUs noch aktualisiert werden oder ob OpenCL durch eine andere API in naher Zukunft de-facto ersetzt wird.

CPU-Unterstützung über einen Kernel-Interpreter: Ein solcher Interpreter würde, wie der Name bereits verrät, ein Kernel via Interpretation mit einer konfigurierbaren Schleifenstruktur ausführen. Leider besitzt eine solche Interpretation verglichen mit der Ausführung von generiertem Code immer einen Overhead. Dieser Overhead ist aber zu vernachlässigen, wenn die Schleifen des Interpreters über genügend viele Komponenten iterieren. Zusätzlich lässt sich dieser Overhead sehr stark reduzieren, indem der Interpreter die Schleifenstrukturen des Kernels nur einmal am Anfang ermittelt, abspeichert und dann bei jeder Ausführung des Kernels lädt. Auf diese Weise muss der Interpreter während der Ausführung eines Kernels keine Berechnungen mehr aufbringen, um die Schleifenstruktur des Kernels zu ermitteln. Des Weiteren kann ein Interpreter nur schlecht Datenwiederverwendung über die Register erzielen. Dies ist kein Problem auf CPUs, da diese hauptsächlich Daten über ihre Caches wiederverwenden, aber ein großes Problem auf GPUs, da hier die Datenwiederverwendung über die Register eine sehr große Rolle spielt. Des Weiteren wäre es für die CPU-Unterstützung unseres Frameworks aufwändiger gewesen, einen Interpreter für CPU-Kernels zu entwickeln als einen Codegenerator für CPU-Kernels zu implementieren.

13.3 Herausforderungen bei den experimentellen Untersuchungen

13.3.1 Workaround für die Untersuchung der Skalierbarkeit mit der Anzahl an benutzten GPU-Kernen

Da man eine moderne GPU nicht so konfigurieren kann, dass sie die Workgroups eines Kernels nicht auf allen ihrer GPU-Kerne, sondern nur auf einem Teil ihrer GPU-Kerne, ausführt, ist es nicht möglich direkt zu evaluieren, wie ein Kernel mit der Zahl der benutzten GPU-Kerne skaliert. Jedoch implementierten wir ein Workaround für dieses Problem. Dieser Workaround reiht zunächst in einer Task-Queue der GPU ein blockierendes Kernel ein, welches für jeden zu blockierenden Kern eine Workgroup startet, die den Kern komplett belegt. Diese Workgroups führen dann nur Dummy-Berechnungen ohne Speicherzugriffe für eine lange Zeit aus. Kurze Zeit später reiht der Workaround in eine andere Task-Queue der GPU das zu untersuchende Kernel ein. Da das blockierende Kernel mittlerweile mit seinen Workgroups die zu blockierenden GPU-Kerne belegt, kann die GPU die Workgroups des zu untersuchenden Kernels nur auf den nicht blockierten Kernen starten. Folglich können wir mit diesem Workaround die Anzahl der für ein Kernel verwendeten GPU-Kerne auf jeden erwünschten Wert reduzieren. Auf der Hawaii-GPU zeigte dieser Workaround zudem, dass die Kerne dieser GPU in Gruppen von vier angeordnet sind, und dass eine solche Gruppe nur Workgroups von demselben Kernel gleichzeitig ausführen kann. Folglich können wir mit diesem Workaround auf der Hawaii-GPU die Skalierung nur für ein Vielfaches von vier Kernen messen.

13.3.2 Probleme bei den Experimenten für Multi-Workgroup-Tiling auf der Polaris-GPU

Auf der Polaris-GPU stießen wir unter der Verwendung von OpenCL bei den Experimenten für das Multi-Workgroup-Tiling auf mehrere Probleme:

- Es kann nach einer atomaren Operation mehrere 100 000 Takte dauern, bis der aktualisierte Wert eines Zählers einer Barriere vom Typ-ATOMIC in den L1-Caches von allen GPU-Kernen sichtbar wird. Jedoch können wir dieses Problem dadurch beheben, dass die Barriere in ihrer Busy-Waiting-Schleife den Zähler nicht direkt über eine gewöhnliche Ladeoperation, sondern über ein atomares Inkrement um null, ausliest, welches die GPU nicht über ihre L1-Caches, sondern über ihren L2-Cache abwickelt.
- Aus unbekannten Gründen erzeugen einige Kernel mit Multi-Workgroup-Tiling auf Polaris einen Fehler oder ein Deadlock, falls sie mehr als zwei Komponenten pro Workitem berechnen. Da unsere aktuelle Autotuning-Funktionalität unseres Frameworks sich nicht von solchen Fehlern oder Deadlocks erholen kann, beschränken wir alle Kernels auf Polaris auf zwei Komponenten pro Workitem.
- Die Anzahl der von einem Kernel benötigten Register kann nicht über die OpenCL-API abgefragt werden, wodurch unser Autotuning-Ansatz mehr Kandidaten für die beste Tile-Breite und Tile-Höhe durchprobieren muss.
- In CUDA kann man für ein Kernel eine maximale Workgroup-Größe und eine minimale Anzahl von Workgroups pro Kern für diese maximale Workgroup-Größe erzwingen. Dahingegen kann man in OpenCL für ein Kernel nur eine maximale Workgroup-Größe, aber keine minimale Anzahl von Workgroups pro Kern für diese maximale Workgroup-Größe,

erzwingen. Da beim Multi-Workgroup-Tiling die GPU sämtliche Workgroups von mindestens einem Tile gleichzeitig ausführen muss, darf unter der Verwendung von OpenCL ein Multi-Workgroup-Tile nicht aus mehr Workgroups bestehen, als die GPU Kerne besitzt. Bedauerlicherweise unterstützt Polaris bis zu 2 560 Workitems pro Kern, aber nur eine maximale Workgroup-Größe von 1 024 Workitems. Dieses Problem in Kombination mit dem Auftreten von Deadlocks bei mehr als zwei zu berechnenden Komponenten pro Workitem reduziert die maximale Tile-Breite auf Polaris.

13.4 Metriken

In dieser Arbeit verwenden wir folgende Metriken für das Profiling unserer Implementierungsvarianten:

- **DRAM-Volumen (V_{DRAM}):** Wir definieren das *DRAM-Volumen* von einem oder mehreren Kernels als die Gesamtzahl an Bytes, die die Kernel zwischen dem Prozessor und dem DRAM transferieren. Dieses DRAM-Volumen berechnen wir über das Produkt der DRAM-Bandbreite w der Kernels und dessen Laufzeit T:

$$V_{\text{DRAM}} = w \cdot T. \tag{13.1}$$

- **Datenvolumen (V_{data}):** Das *Datenvolumen* soll eine theoretische untere Grenze für das DRAM-Volumen sein. Dabei nimmt das Datenvolumen keine Wiederverwendung von Vektorkomponenten über die Caches des Prozessors zwischen zwei aufeinanderfolgenden Kernels an. Zusätzlich nimmt es an, dass innerhalb eines Kernels mehrmalige Zugriffe auf dieselbe Vektorkomponente durch die Caches des Prozessors abgefangen werden. Je nach Variante berechnet sich das Datenvolumen unterschiedlich:
  - **Datenvolumen der Varianten der Basisimplementierung und der Varianten mit Kernelfusion:** Da für die Varianten der Basisimplementierung und die Varianten mit Kernelfusion unsere Kernels nur, abgesehen von den zeitvarianten Koeffizienten der Konstantengenerierung und den Skalaren der Reduktion, komplette permanente Vektoren aus dem DRAM lesen oder in den DRAM schreiben, können wir hierfür das Datenvolumen berechnen als:

$$V_{\text{data}} = v \cdot n \cdot \text{sizeof(precision)}. \tag{13.2}$$

    Dabei ist n die Dimension des ODE-Systems, sizeof(precision) die Anzahl an Bytes per Vektorelement und v die Anzahl von permanenten Vektoren, die von dem Kernel gelesen oder zurückgeschrieben werden, wie wir sie bei den Varianten mit Kernelfusion in den Tabellen 9.5, 9.8, 9.10 und 9.12 berechneten.
  - **Datenvolumen der Varianten mit Tiling:** Bei den Varianten mit Tiling liest und schreibt ein Kernel nicht nur komplette permanente Vektoren aus dem DRAM beziehungsweise in den DRAM, sondern für einige permanente Vektoren liest und schreibt ein Kernel auch nur Blöcke von aufeinanderfolgenden Komponenten mit einem festen Stride untereinander aus den DRAM beziehungsweise in den DRAM. Deshalb berechnet sich bei den Varianten mit Tiling das Datenvolumen als:

$$V_{\text{data}} = \sum_{i=1}^{v} k_i \cdot \text{sizeof(precision)}. \tag{13.3}$$

    Dabei ist k_i die Anzahl an Komponenten, die das Kernel mit Tiling von dem $i$ten Argumentvektor aus dem DRAM liest oder in den DRAM schreibt, sizeof(precision) ist die Anzahl an Bytes per Vektorelement und v ist die Anzahl an Argumentvektoren des Kernels.

- **Cache-Effizienz (E_{cache}):** Die *Cache-Effizienz* beschreibt, wie effizient die Caches das DRAM-Volumen reduzieren können, oder genauer, wie nahe das DRAM-Volumen beim Datenvolumen ist. Folglich definieren wir die Cache-Effizienz von einem oder mehreren Kernels als das Verhältnis zwischen deren Datenvolumen und deren DRAM-Volumen:

$$E_{\text{cache}} = \frac{V_{\text{data}}}{V_{\text{DRAM}}}. \tag{13.4}$$

  Dabei kann jedoch eine eigentlich ausgeschlossene Datenwiederverwendung zwischen zwei aufeinanderfolgenden Kernels oder auch Ungenauigkeiten beim Profiling des DRAM-Volumens eine Cache-Effizienz von etwas über 100 % verursachen.

- **Auslastung der DRAM-Bandbreite (U_{DRAM}):** Wir definieren die *Auslastung der DRAM-Bandbreite* als das Verhältnis der im Profiler gemessenen DRAM-Bandbreite w zu seiner Burst-DRAM-Bandbreite w_{Burst}

$$U_{\text{DRAM}} = \frac{w}{w_{\text{Burst}}} \leq 1. \tag{13.5}$$

  Hierbei gilt es zu beachten, dass der DRAM seine Burst-DRAM-Bandbreite auf Grund von DRAM-Refreshes und Pipeline-Stalls beziehungsweise Pipeline-Bubbles nicht über einen längeren Zeitraum aufrechterhalten kann. Gemäß unseren Erfahrungen kann deshalb ein Kernel, selbst wenn es durch die DRAM-Bandbreite gebunden ist, nur, je nach seinem Speicherzugriffsmuster, der Architektur des Prozessors und der verwendeten DRAM-Technologie, ungefähr 70 % bis 95 % der Burst-DRAM-Bandbreite ausnutzen. Eine Behandlung und eine experimentelle Untersuchung dieser Problematik kann zum Beispiel in der Arbeit [K38] gefunden werden. In der Literatur wird die Burst-DRAM-Bandbreite auch oft als potentielle Bandbreite, wie zum Beispiel in [T28], als maximale Bandbreite oder als Peak-Bandbreite, wie zum Beispiel in [K38], bezeichnet. Wir verwenden in dieser Arbeit jedoch den Begriff der Burst-DRAM-Bandbreite, da dieser Begriff die Natur dieser Bandbreite, und zwar dass der DRAM sie nur für einen kurzen Zeitraum aufrechterhalten kann, am besten beschreibt.

- **Anzahl an herausgegebenen Instruktionen (INS):** Die Metrik der *Anzahl an herausgegebenen Instruktionen* beschreibt, wie viele Instruktionen die Scheduler der Kerne insgesamt während der Ausführung eines Kernels an die Ausführungseinheiten herausgeben.

- **Anzahl an herausgegebene Instruktionen pro Takt (IPC):** Die Metrik der *Anzahl an herausgegebenen Instruktionen pro Takt* beschreibt, wie viele Instruktionen pro Takt die Scheduler der Kerne im Durchschnitt an die Ausführungeinheiten während der Ausführung eines Kernels herausgeben.

- **Auslastung der IPC (U_{IPC}):** Wir definieren die *Auslastung der ICP* für ein Kernel als dasjenige Verhältnis zwischen den tatsächlich herausgegebenen Instruktionen pro Takt (IPC) zu derjenigen Anzahl an Instruktionen pro Takt (IPC_{max}), die ein Kern des Prozessors maximal herausgeben kann:

$$U_{\text{IPC}} = \frac{\text{IPC}}{\text{IPC}_{\text{max}}} \leq 1. \tag{13.6}$$

  Wir können die Auslastung der IPC dafür verwenden, um auf GPUs für Kernels mit der einfachen Gleitkommagenauigkeit die Auslastung der SP-FPU+INT32-ALUs abzuschätzen. Dabei gilt es zu beachten, dass zum Beispiel ein Kern auf einer GPU der

Maxwell Architektur bis zu 8 Instruktionen pro Takt herausgeben kann, während seine SP-FPU+INT32-ALUs nur bis zu 4 Instruktionen pro Takt verarbeiten können. Ebenso kann ein Kern der Kepler-Architektur bis zu 8 Instruktionen pro Takt herausgeben, während seine SP-FPU+INT32-ALUs nur bis zu 6 Instruktionen pro Takt verarbeiten können. Folglich kann ein Kernel auf einer GPU mit der Maxwell-Architektur bereits durch dessen Rechenleistung gebunden sein, falls die Anzahl an herausgegebenen Instruktionen pro Takt mehr als 50 % von IPC_{max} beträgt, und auf einer GPU mit der Kepler-Architektur, falls die Anzahl an herausgegebenen Instruktionen pro Takt mehr als 75 % von IPC_{max} beträgt.

- **Auslastung der DP-Rechenwerke (U_{DP}):** Wir definieren die *Auslastung der DP-Rechenwerke* als dasjenige Verhältnis von den durch einen Prozessor herausgegebenen DP-Instruktionen pro Takt (IPC_{DP}) zu der maximalen Anzahl an DP-Instruktionen pro Takt ($\text{IPC}_{\text{DP, max}}$), die die DP-Rechenwerke des Prozessors maximal verarbeiten können:

$$U_{\text{DP}} = \frac{\text{IPC}_{\text{DP}}}{\text{IPC}_{\text{DP, max}}} \leq 1. \tag{13.7}$$

  Hierbei gilt anzumerken, dass die Metrik der Auslastung der DP-Rechenwerke Fused-Multiply-Add-Instruktionen, Multiplikationsinstruktionen und Additionsinstruktionen je nur einfach gewichtet. Zudem berücksichtigt diese Metrik nicht, dass auf einer SIMD-Architektur in einem Kernel durch Predication oder schlechte Vektorisierung SIMD-Lanes in einem DP-Rechenwerk ungenutzt sein können, wodurch wiederum Rechenleistung verloren gehen kann. Deshalb eignet sich diese Metrik dafür sehr gut, um zu erkennen, inwieweit die DP-Rechenwerke in einem Kernel ein Flaschenhals darstellen. Sie eignet sich aber nicht, um zu erkennen, wie viel DP-Rechenleistung ein Prozessor für ein Kernel tatsächlich erzielen kann.

- **Ausnutzung der DP-Peak-Performance ($U_{\text{DP-PEAK}}$):** Wir definieren die *Ausnutzung der DP-Peak-Performance* als das Verhältnis von denjenigen DP-FLOPS, die ein Kernel auf einem Prozessor erzielt, zu den maximalen DP-FLOPS des Prozessors, wobei wir, wie bei dieser Metrik allgemein üblich, Add-DP-FLOPS sowie Mul-DP-FLOPS einfach und FMA-DP-FLOPS doppelt gewichten:

$$U_{\text{DP-PEAK}} = \frac{\text{FLOPS}_{\text{DP, Add}} + \text{FLOPS}_{\text{DP, Mul}} + 2 \cdot \text{FLOPS}_{\text{DP, FMA}}}{\text{FLOPS}_{\text{DP, Add, max}} + \text{FLOPS}_{\text{DP, Mul, max}} + 2 \cdot \text{FLOPS}_{\text{DP, FMA, max}}} \leq 1. \tag{13.8}$$

  Dadurch eignet sich die Metrik der Ausnutzung der DP-Peak-Performance im Gegensatz zur Metrik der Auslastung der DP-Rechenwerke nicht, um festzustellen, ob die DP-Rechenwerke in einem Kernel zu einem Flaschenhals werden. Falls ein bestimmter Prozessor nicht all seine DP-Rechenwerke gleichzeitig auslasten kann, oder falls ein DP-Rechenwerk mehrere unterschiedliche Arten von Gleitkommaoperationen durchführen kann, muss für diesen Prozessor die Formel zur Berechnung von dessen maximalen DP-FLOPS angepasst werden.

13.5 GPU-Daten

Dieser folgende Abschnitt gibt eine Übersicht über die technischen Daten und die Roh-Performance mehrerer moderner GPUs aus den Jahren 2013 bis 2020. Dabei stammen diese technischen Daten aus den Whitepapers und ISA-Dokumentationen der GPU-Architekturen sowie aus den Programming-Guides von NVIDIA [T2–T9], AMD [T10–T15] und Intel [T16–T19]. In dieser Arbeit geben wir die technischen Daten von folgenden GPUs an:

- Tabelle 13.1 zeigt die technischen Daten dreier NVIDIA-GPUs:
  - Titan Black aus dem Jahr 2014
  - Geforce 980 Ti aus dem Jahr 2015
  - Tesla P100 aus dem Jahr 2016
- Tabelle 13.2 zeigt die technischen Daten dreier NVIDIA-GPUs:
  - Titan V aus dem Jahr 2017
  - Geforce RTX 3090 aus dem Jahr 2020
  - A100 aus dem Jahr 2020
- Tabelle 13.3 zeigt die technischen Daten dreier AMD-GPUs:
  - Firepro W8100 aus dem Jahr 2014
  - Radeon RX 480 aus dem Jahr 2016
  - Radeon RX 6900 aus dem Jahr 2020
- Tabelle 13.4 zeigt die technischen Daten dreier Intel-GPUs:
  - HD 4600 aus dem Jahr 2012
  - Iris Plus 650 aus dem Jahr 2015
  - Xe DG1 aus dem Jahr 2020

Zu diesen Tabellen ist Folgendes anzumerken:

- Die Kepler-Architektur von NVIDIA und sämtliche GPU-Architekturen von Intel besitzen noch spezialisierte Textur-Caches, die während eines Kernels nicht kohärent gehalten werden und die ein Kernel auch für gewöhnliche Gleitkommadaten verwenden kann. Wir fügten diese Textur-Caches in der Tabelle aus Platzgründen nicht mehr ein.

- Der L1-Cache auf der Kepler Titan Black kann nur Zugriffe auf den Stack-Speicher zwischenspeichern. Ursache hierfür ist wahrscheinlich ein Erratum bei der Cache-Kohärenz der GPU. Diese Vermutung wird dadurch gestützt, dass in einer späteren Revision der Kepler-Architektur diese Einschränkung nicht mehr existiert.

- Auf mehreren NVIDA-GPUs teilen sich der L1-Cache und der Scratchpad-Speicher denselben physikalischen Speicher, wobei die Aufteilung zwischen beiden konfigurierbar ist. Wir versahen deshalb bei solchen GPUs in der Tabelle die Größe des L1-Caches und des Scratchpad-Speichers mit dem Term "bis zu".

- Welche Vektorrechenwerke ein Subkern besitzt, welche Instruktionen ein Vektorrechenwerk ausführen kann, und ob ein Vektorrechenwerk zwischen den Subkernen geteilt ist, wird leider von den GPU-Herstellern oft nur sehr ungenau oder fehlerhaft dokumentiert. Aus diesem Grund können diese Tabelleneinträge fehlerhaft sein und sollten deshalb nur zur Veranschaulichung herangezogen werden.

Die Rohperformance der GPUs aus den Tabellen 13.1 13.2 13.3 und 13.4 wird in der Tabelle 13.5 gezeigt. Zu dieser Tabelle ist Folgendes anzumerken:

- Moderne GPUs besitzen eine Boost-Funktion, mit Hilfe derer sie ihren Takt erhöhen können, sofern dies ihre Temperatur und ihr Stromverbrauch zulässt. Jedoch können moderne GPUs ihren erhöhten Boost-Takt gerade bei sehr rechenintensiven Kernels nur für eine kurze Zeit beibehalten, da gerade bei solchen Kernels eine GPU viel Strom verbraucht und sich deshalb stark erwärmt. Deshalb gibt die Tabelle 13.5 nur den GPU-Basistakt ohne die Boost-Erhöhung und die daraus resultierende Peak-Gleitkomma-Performance an.

- Diese Tabelle gibt, wie allgemein üblich, die Taktung der DRAM-Chips indirekt über die Transferrate einer Datenleitung in Bit-Tranfers pro Sekunde [T/s] an. Bei früheren DRAM-Technologien wie SDRAM war die Transferrate eines DRAM-Chips noch identisch zu dessen Takt. Bei moderneren DRAM-Technologien überträgt auf Grund von Double-Data-Rate und Prefetching ein DRAM-Chip pro Takt mehrere Bits über eine Datenleitung.

- Bei der angegebenen DRAM-Bandbreite handelt es sich um die sogenannte Burst-DRAM-Bandbreite, welche die GPU nur für einen sehr kurzen Zeitraum erreichen kann. Ursachen hierfür sind DRAM-Refreshes, Bank-Konflikte, Stalls beziehungsweise Bubbles in der Pipeline des entsprechenden DRAM-Protokolls und Latenzen. Der Einfluss der DRAM-Refreshes auf die tatsächlich erreichbare DRAM-Bandbreite unterscheidet sich je nach DRAM-Technologie (für eine ausgiebigere Erläuterung siehe zum Beispiel [T28]):

  - **DRAM-Refreshes bei der GDDR-Technologie:** Eine GPU mit GDDR-Speicher kann nur sämtliche Speicherbänke eines GDDR-Chips gleichzeitig auffrischen. Dieses Auffrischen verursacht wiederum einen Stall in der Pipeline des GDDR-Protokolls. Deshalb kann ein GDDR-Chip nicht nur während des Auffrischens keine Daten an die GPU schicken, sondern nach dem Auffrischen muss die GPU die Pipeline des GDDR-Protokolls neu initialisieren. Dadurch dauert es nach dem Auffrischen noch mehrere Takte, bis der GDDR-Chip die nächsten Daten zur GPU schickt. Bis dahin bleibt der Datenbus des GDDR-Chips unbenutzt. Dieser Refresh führt dazu, dass GDDR-Speicher nur 90 % bis 95 % seiner Burst-Bandbreite über einen längeren Zeitraum ausnutzen kann.
  - **DRAM-Refreshes bei der HBM-Technologie:** Im Gegensatz zu einer GPU mit GDDR-Speicher kann eine GPU mit HBM-Speicher jedoch auch nur eine einzelne Speicherbank in einem HBM-Chip auffrischen und die übrigen Speicherbänke des HBM-Chips dafür verwenden, um den Datenbus des HBM-Chips während des Auffrischens dieser einzelnen Bank weiterhin zu saturieren. Hierdurch wird der Overhead der DRAM-Refreshes fast vollkommen negiert.

GPU	Titan Black	GeForce 980 Ti	Tesla P100
Architektur	Kepler	Maxwell	Pascal (Compute)
Jahr	2014	2015	2016
Speicher			
Typ	GDDR5	GDDR5	HBM2
Ausbau	6 GiB	6 GiB	16 GiB
Interface	384 Bit	384 Bit	4096 Bit
Caches			
L1	15 × 48 kiB (bis zu, privat je Kern)	22 × 48 kiB (privat je Kern)	56 × 24 kiB (privat je Kern)
L2	3 MiB (geteilt)	3 MiB (geteilt)	4 MiB (geteilt)
Kerne			
# Kerne	15	22	56
Scratchpadspeicher pro Kern	48 kiB (bis zu)	96 kiB	64 KiB
# Subkerne pro Kern	4	4	2
Max. # Threads pro Subkern	16	16	32
Kontrollvektorbreite	32	32	32
# Vektorregister pro Subkern	512	512	1024
Aufbau eines Vektorregisters	32 × 32-Bit Lanes	32 × 32-Bit Lanes	32 × 32-Bit Lanes
Vektorrechenwerke pro Subkern (mit Vektorbreite)	SP-FPU (32×)+ INT32-ALU (32×), DP-FPU (16×)	SP-FPU (32×)+ INT32-ALU (32×), DP-FPU (2×)	SP-FPU (32×)+ INT32-ALU (32×), DP-FPU (16×), SP-SFU (8×)
Vektorrechenwerke geteilt zwischen je zwei Subkernen (mit Vektorbreite)	SP-FPU (32×)+ INT32-ALU (32×), SP-FPU (16×)	SP-SFU (4×)	

Tabelle 13.1: Technische Daten dreier NVIDIA GPUs aus den Jahren 2014 bis 2016.

GPU	**Titan V**	**Geforce RTX 3090**	**A100**
Architektur	Volta	Ampere	Ampere (Compute)
Jahr	2017	2020	2020
Speicher			
Typ	HBM2	GDDR6X	HBM2
Ausbau	10 GiB	24 GiB	80 GiB
Interface	3072 Bit	384 Bit	5120 Bit
Caches			
L1	80 × 128 kiB (bis (zu, privat je Kern)	82 × 128 kiB (bis zu, privat je Kern)	108 × 128 kiB (bis zu, privat je Kern)
L2	6 MiB (geteilt)	6 MiB (geteilt)	40 MiB (geteilt)
Kerne			
# Kerne	80	82	108
Scratchpadspeicher pro Kern	96 kiB (bis zu)	100 kiB (bis zu)	164 KiB (bis zu)
# Subkerne pro Kern	4	4	4
Max. # Threads pro Subkern	16	12	16
Kontrollvektorbreite	32	32	32
# Vektorregister pro Subkern	512	512	512
Aufbau eines Vektorregisters	32 × 32-Bit Lanes	32 × 32-Bit Lanes	32 × 32-Bit Lanes
Vektorrechenwerke pro Subkern (mit Vektorbreite)	SP-FPU (16×), INT32-ALU (16×), DP-FPU (8×), SP-SFU (4×), 2× Tensor-FPU (64×)	SP-FPU (16×)+ INT32-ALU (16×), SP-FPU (16×), SP-SFU (4×), Tensor-FPU (256×)	SP-FPU (16×), INT32-ALU (16×), DP-FPU (8×), SP-SFU (4×), Tensor-FPU (256×)
Vektorrechenwerke geteilt zwischen je zwei Subkernen (mit Vektorbreite)		DP-FPU (1×)	

Tabelle 13.2: Technische Daten dreier NVIDIA GPUs aus den Jahren 2017 bis 2020

GPU	Firepro W8100	Radeon RX 480	Radeon RX 6900
Architektur	GCN 2 (Hawaii)	GCN 4 (Polaris)	RDNA 2 (Navi)
Jahr	2014	2016	2020
Speicher			
Typ	GDDR5	GDDR5	GDDR6
Ausbau	8 GiB	8 GiB	16 GiB
Interface	512 Bit	256 Bit	256 Bit
Caches			
L1	40 × 16 kiB (privat je Kern)	32 × 16 kiB (privat je Kern)	80 × 64 kiB (privat je Kern)
L2	1 MiB (geteilt)	2 MiB (geteilt)	16 × 128 kiB (gtlt. je 5 Kerne)
L3	-	-	4 MiByte (geteilt)
L4	-	-	128 MiByte (geteilt)
Kerne			
# Kerne	40	32	80
Scratchpadspeicher pro Kern	64 kiB	64 kiB	128 KiB
# Subkerne pro Kern	4	4	4
Max. # Threads pro Subkern	10	10	20
Kontrollvektorbreite	64	64	32 oder 64
# Vektorregister pro Subkern	512	512	1024
Aufbau eines Vektorregisters	64 × 32-Bit Lanes	64 × 32-Bit Lanes	32 × 32-Bit Lanes
Vektorrechenwerke pro Subkern (mit Vektorbreite)	SP-FPU (16×)+ INT32-ALU (16×)+ DP-FPU (8×)	SP-FPU (16×)+ INT32-ALU (16×)+ DP-FPU (1×)	SP-FPU (32×)+ INT32-ALU (32×), DP-FPU (2×), SP-SFU (8×)

Tabelle 13.3: Technische Daten dreier AMD GPUs aus den Jahren 2014 bis 2020.

GPU	HD 4600	Iris Plus 650	Xe DG1
CPU	Core i7-4790	Core i7-8569U	n/a (diskrete GPU)
Architektur	Proc. Grphc. Gen7	Proc. Grphc. Gen9	?
Jahr	2013	2017	2020
Speicher			
Typ	DDR3	DDR4	DDR4
Ausbau	modular	modular	4 GiB
Interface	128 Bit	128 Bit	256 Bit
Caches			
L1	256 kiB (privat für die gesamte GPU)	2 × 512 kiB (prvt. für je drei GPU-Kerne)	?
L2	8 MiB (geteilt mit CPU als L3-Cache)	8 MiB (geteilt mit CPU als L3-Cache)	16 MiB (geteilt)
L3	-	128 MiB (geteilt mit CPU als L4-Cache)	-
Kerne			
# Kerne	2	6	6
Scratchpadspeicher pro Kern	64 kiB	64 kiB	?
# Subkerne pro Kern	10	8	16
# Threads pro Subkern	7	7	7
Kontrollvektorbreite	1, 2, 4, 8, 16 oder 32	1, 2, 4, 8, 16 oder 32	?
# Vektorregister pro Thread	128	128	?
Aufbau eines Vektorregisters	8 × 32-Bit Lanes	8 × 32-Bit Lanes	?
Vektorrechenwerke pro Subkern (mit Vektorbreite)	SP-FPU (4×)+ INT32-ALU (4×), SP-FPU (4×)+ DP-FPU (2×, 2 Takte pro Instruktion)	SP-FPU (4×)+ INT32-ALU (4×), SP-FPU (4×)+ INT32-ALU(4×)+ DP-FPU (2×, 2 Takte pro Instruktion)	?

Tabelle 13.4: Technische Daten dreier Intel GPUs aus den Jahren 2013 bis 2020.

NVIDIA GPUs	GPU-Takt in [MHz]	SP-Performance in [TFLOP/s]	DP-Performance in [TFLOP/s]	DRAM-Transferrate in [GT/s]	DRAM-Bandbr. in [GB/s]	Bytes per SP/DP-FLOP
Titan Black	889	5.1	1.7	7.0	337	0.065 / 0.196
GFrc. 980 Ti	1 024	5.7	0.2	7.0	337	0.059 / 1.685
Tesla P100	1 126	8.4	4.2	1.4	732	0.087 / 0.174
Titan V	1 200	12.3	6.1	1.7	853	0.069 / 0.139
GFrc. 3090	1 395	29.3	0.46	21.0	936	0.032 / 2.034
A100	765	12.4	6.2	1.8	1 555	0.125 / 0.251

AMD GPUs	GPU-Takt in [MHz]	SP-Performance in [TFLOP/s]	DP-Performance in [TFLOP/s]	DRAM-Transferrate in [GT/s]	DRAM-Bandbr. in [GB/s]	Bytes per SP/DP-FLOP
Firepro W8100	824	4.2	2.1	5.0	320	0.076 / 0.152
Rad. RX 480	1 120	5.2	0.3	7.0	224	0.043 / 0.747
Rad. RX 6900	1 825	18.7	1.2	16.0	512	0.027 / 0.43

Intel GPUs	GPU-Takt in [MHz]	SP-Performance in [TFLOP/s]	DP-Performance in [TFLOP/s]	DRAM-Transferrate in [GT/s]	DRAM-Bandbr. in [GB/s]	Bytes per SP/DP-FLOP
HD Grph. 4600	1 250	0.4	0.1	1.6	26	0.065 / 0.260
Iris Plus 650	1 150	0.9	0.2	2.4	38	0.042 / 0.169
Xe DG1	1 650	2.5	-	4.3	68	0.027 / -

Tabelle 13.5: Rohperformance der GPUs aus den Jahren 2013 bis 2020.

13.6 CPU-Daten

Dieser folgende Abschnitt gibt eine Übersicht über die technischen Daten und die Roh-Performance mehrerer moderner x64-CPUs von Intel und AMD aus dem Desktop und Serverbereich, welche in den Jahren 2012 bis 2021 eingeführt wurden. Die technischen Daten stammen hierbei aus dem Kompendium zur Mikroarchitektur von modernen Intel, AMD und VIA-CPUs von Agner Fog [T22], welches als Standardliteratur zu diesem Thema betrachtet werden kann, und von der Internetenzyklopädie Wikichip [W15]. In dieser Arbeit geben wir die technischen Daten von folgenden GPUs an:

- Tabelle 13.6 zeigt die technischen Daten dreier Intel-CPUs:
  - Core i7 4790K aus dem Jahr 2012
  - Core i9 9980XE aus dem Jahr 2018
  - Xeon Platinum 8280 aus dem Jahr 2019
- Tabelle 13.7 zeigt die technischen Daten dreier dreier AMD-CPUs:
  - FX 9590 aus dem Jahr 2013
  - Ryzen 5950 aus dem Jahr 2020
  - EPYC 7763 aus dem Jahr 2021

Zu diesen Tabellen ist Folgendes anzumerken:

- Moderne x64-CPUs besitzen neben Vektor-FPUS, Vektor-ALUs und Vektorregister auch skalare ALUs und skalare Register. Dabei bezeichnen AMD und Intel die skalaren Register oft als Integer-Register. Zusätzlich bezeichnet AMD die Vektorregister als Floating-Point-Register und den Aufühungspfad für Vektor-Instruktionen als Floating-Point-Execution. Da diese AMD-CPUs innerhalb der Floating-Point-Execution auch Vektor-ALUs besitzen, die auf mit Integern befüllten Vektorregistern arbeiten, ist diese Bezeichnung unpassend.
- AMD CPUs mit der Zen3-Architektur besitzen vier Vektor-FPUs, von denen die einen beiden FMA-Instruktionen, Mul-Instruktionen sowie Add-Instruktionen ausführen können, während die anderen beiden nur Add-Instruktionen ausführen können.
- Auf Grund von Register-Renaming besitzt eine moderne x64-CPU mehr physikalische Vektorregister als Architekturvektorregister in ihrem Befehlssatz vorhanden sind. So definiert zum Beispiel die AVX-512 SIMD-Befehlssatzerweiterung für x64-CPUs nur 32 Architekturvektorregister mit je 512 Bit, während ein Kern einer Xeon Platinum 8280 CPU auf Grund von Register-Renaming 168 physikalische Vektorregister mit je 512 Bit besitzt. Ein Programm kann allerdings diese physikalischen Register nur indirekt über die geschickte Ausnutzung der Out-Of-Order-Pipeline der CPU verwenden.

Die Roh-Performance der CPUs wird in der Tabelle 13.8 gezeigt. Zu dieser Tabelle ist Folgendes anzumerken:

- Moderne CPUs besitzen eine Boost-Funktion, mit Hilfe derer sie ihren Takt von einem Basistakt auf einen Boost-Takt erhöhen können, sofern dies ihre Temperatur und ihr Stromverbrauch zulassen. Jedoch können moderne CPUs ihren erhöhten Boost-Takt

gerade bei sehr rechenintensiven Programmen nur für eine kurze Zeit halten, da gerade bei solchen Programmen eine CPU viel Strom verbraucht und sie sich deshalb stark erwärmt. Zusätzlich gilt auf modernen Intel-CPUs mit AVX-512, dass wenn ein Programm auf vielen Kernen viele solcher AVX-512-Instruktionen ausführt, sich die CPU von ihrem Basistakt auf einen AVX-512-Takt herabtaktet. Deshalb gibt die Tabelle 13.5 nur den Basistakt der CPUs ohne die Boost-Erhöhung beziehungsweise den AVX-512-Takt und die daraus resultierende Peak-Gleitkomma-Performance an.

- Wie bereits erwähnt, so besitzen AMD-CPUs mit der Zen3-Architektur vier Vektor-FPUs, von denen die einen beiden Add-Instruktionen, Mul-Instruktionen sowie FMA-Instruktionen und die anderen beiden nur Add-Instruktionen ausführen können. Allerdings werden aus unbekannten Gründen von [W15] nur die Vektor-FPUs mit den FMA-Instruktionen und nicht zusätzlich die Vektor-FPUs mit den ADD-Instruktionen zur Berechnung der Peak-Performance herangezogen. In dieser Arbeit berechnen wir die Peak-Performance von Zen3-CPUs analog zu [W15].

- Bei der angegebenen DRAM-Transferrate handelt es sich um die maximale Transferrate, für die der DRAM-Contoller der CPU spezifiziert ist. Durch das Verbauen von DRAM mit einer höheren Transferrate betreibt man somit den DRAM-Contoller der CPU außerhalb seiner Spezifikation. De facto lässt sich auf den meisten CPUs aber DRAM mit einer Transferrate betreiben, welche deutlich größer als die spezifizierte maximale Transferrate des DRAM-Contollers ist.

- Bei der angegebenen DRAM-Bandbreite handelt es sich um die sogenannte Burst-Bandbreite, welche der DRAM nur für einen sehr kurzen Zeitraum aufrechterhalten kann. Ursachen hierfür sind DRAM-Refreshes, Bank-Konflikte, Stalls beziehungsweise Bubbles in der Pipeline des entsprechenden DRAM-Protokolls und Latenzen.

CPU	**Core i7 4790K**	**Core i9 9980XE**	**Xeon Plat. 8280**
Architektur	Haswell	Skylake	Cascade Lake
Jahr	2012	2018	2019
Speicher			
Typ	DDR3	DDR4	DDR4
Interface	128	256	384
Caches			
L1	4 × 32 KiB (privat je Kern)	18 × 32 KiB (privat je Kern)	28 × 32 KiB (privat je Kern)
L2	4 × 256 KiB (privat je Kern)	18 × 1 MiB (privat je Kern)	28 × 1 MiB (privat je Kern)
L3	8 MiB (geteilt)	24.75 MiB (geteilt)	38.5 MiB (geteilt)
Kerne			
# Kerne	4	18	28
# Subkerne pro Kern	1	1	1
# Threads pro Subkern	2	2	2
Logische Vektorbreite	256 Bit	512 Bit	512 Bit
# Physikal. Vektorregister pro Subkern	160 × 256-Bit-Register	168 × 512-Bit-Register	168 × 512-Bit-Register
Vektorrechenwerke pro Subkern (mit Vektorbreite)	2 × ALU+FPU (256 Bit), 1 × ALU (256 Bit)	2 × ALU+FPU (256 Bit, verschmelzbar zu 512 Bit), 1 × ALU+FPU (512 Bit)	2 × ALU+FPU (256 Bit, verschmelzbar zu 512 Bit), 1 × ALU+FPU (512 Bit)

Tabelle 13.6: Technische Daten dreier Intel CPUs aus den Jahren 2012 bis 2019.

CPU	**FX 9590**	**Ryzen 5950**	**EPYC 7763**
Architektur	Piledriver	Zen3	Zen3
Jahr	2013	2020	2021
Speicher			
Typ	DDR3	DDR4	DDR4
Interface	128 Bit	128 Bit	512 Bit
Caches			
L1	8 × 16 kiB (privat je Subkern)	16 × 32 kiB (privat je Kern)	64 × 32 kiB (privat je Kern)
L2	4 × 2 MiB (privat je Kern)	16 × 256 kiB (privat je Kern)	64 × 256 kiB (privat je Kern)
L3	8 MiB (geteilt)	2 × 32 MiB (privat je 8 Kerne)	8 × 32 MiB (privat je 8 Kerne)
Kerne			
# Kerne	4	16	64
# Subkerne pro Kern	2	1	1
Max. # Threads pro Subkern	1	2	2
Logische Vektor-breite	128 Bit	256 Bit	256 Bit
# Physikal. Vektor-register pro Subkern	n/a (nur skalare Register)	160 × 256-Bit-Register	160 × 256-Bit-Register
# Phys. Vekregs. getlt. zw. Subkernen	160 × 128-Bit-Register		
Vektorrechen-werke pro Subkern (mit Vektorbreite)	n/a (nur skalare ALUs)	4 × ALU+FPU (256 Bit)	4 × ALU+FPU (256 Bit)
Vektorrechen-werke geteilt zwischen Subkernen (mit Vektorbreite)	2 × ALU (128 Bit), 2 × FPU (128 Bit)		

Tabelle 13.7: Technische Daten dreier AMD CPUs aus den Jahren 2013 bis 2021.

Intel CPUs	CPU-Takt in [MHz]	SP-Perf. in [TFLOP/s]	DP-Perf. in [TFLOP/s]	DRAM-Transferrate in [GT/s]	DRAM-Bandbr. in [GB/s]	Bytes per SP/DP-FLOP
Core i7 4790K	4 000	0.51	0.26	1.60	25.6	0.050 / 0.100
Core i9 9980XE	2 800	3.23	1.61	2.67	85.2	0.026 / 0.053
Xeon Plat. 8280	2 700	4.84	2.42	2.93	141.0	0.029 / 0.058

AMD CPUs	CPU-Takt in [MHZ]	SP-Perf. in [TFLOP/s]	DP-Perf. in [TFLOP/s]	DRAM-Transferrate in [GT/s]	DRAM-Bandbr. in [GB/s]	Bytes per SP/DP-FLOP
FX 9590	4 700	0.30	0.15	2.4	38.4	0.128 / 0.256
Ryzen 5950	3 400	1.74	0.87	3.2	51.2	0.029 / 0.059
EPYC 7763	2 450	5.02	2.51	3.2	204.8	0.041 / 0.082

Tabelle 13.8: Rohperformance der CPUs aus den Jahren von 2013 bis 2020.

13.7 Verwendete RK-Verfahren

Folgende RK-Verfahren wurden in dieser Arbeit verwendet:

- **Explizites Eulerverfahren [B7]:**

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array}$$

- **Implizites Eulerverfahren [B7]:**

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$$

- **Midpoint-Verfahren [B7]:**

$$\begin{array}{c|cc} 0 & & \\ \frac{1}{2} & \frac{1}{2} & \\ \hline & 0 & 1 \end{array}$$

- **Heun-Verfahren [B7]:**

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

- **Heun-Euler-Verfahren [B7]:**

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & \frac{1}{2} & \frac{1}{2} \\ & 1 & 0 \end{array}$$

- **RK-3/8-Verfahren [B7]:**

$$
\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{3} & \frac{1}{2} & & & \\
\frac{2}{3} & -\frac{1}{3} & 1 & & \\
1 & 1 & -1 & 1 & \\
\hline
 & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8}
\end{array}
$$

- **Bogacki–Shampine-Verfahren [Z45]:**

$$
\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & & \\
\frac{3}{4} & 0 & \frac{3}{4} & & \\
1 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & \\
\hline
 & \frac{2}{9} & \frac{1}{3} & \frac{4}{9} & 0 \\
 & \frac{7}{24} & \frac{1}{4} & \frac{1}{3} & \frac{1}{8}
\end{array}
$$

- **Lobatto-IIIC-Verfahren [Z46]:**

$$
\begin{array}{c|ccccc}
0 & \frac{1}{20} & -\frac{7}{60} & \frac{2}{15} & -\frac{7}{60} & \frac{1}{20} \\
\frac{1}{2}-\frac{\sqrt{21}}{14} & \frac{1}{20} & \frac{29}{180} & \frac{47-15\sqrt{21}}{315} & \frac{203-30\sqrt{21}}{1260} & -\frac{3}{140} \\
\frac{1}{2} & \frac{1}{20} & \frac{329+105\sqrt{21}}{2880} & \frac{73}{360} & \frac{329-105\sqrt{21}}{2880} & \frac{3}{160} \\
\frac{1}{2}+\frac{\sqrt{21}}{14} & \frac{1}{20} & \frac{203+30\sqrt{21}}{1260} & \frac{47+15\sqrt{21}}{315} & \frac{29}{180} & -\frac{3}{140} \\
1 & \frac{1}{20} & \frac{49}{180} & \frac{16}{45} & \frac{49}{180} & \frac{1}{20} \\
\hline
 & \frac{1}{20} & \frac{49}{180} & \frac{16}{45} & \frac{49}{180} & \frac{1}{20}
\end{array}
$$

- **DOPRI-5-Verfahren [Z47]:**

$$
\begin{array}{c|ccccccc}
0 & & & & & & & \\
\frac{1}{5} & \frac{1}{5} & & & & & & \\
\frac{3}{10} & \frac{3}{40} & \frac{9}{40} & & & & & \\
\frac{4}{5} & \frac{44}{45} & -\frac{56}{15} & \frac{32}{9} & & & & \\
\frac{8}{9} & \frac{19372}{6561} & -\frac{25360}{2187} & \frac{64448}{6561} & -\frac{212}{729} & & & \\
1 & \frac{9017}{3168} & -\frac{355}{33} & \frac{46732}{5247} & \frac{49}{176} & -\frac{5103}{18656} & & \\
1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & \\
\hline
 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & -\frac{2187}{6784} & \frac{11}{84} & 0 \\
 & \frac{5179}{57600} & 0 & \frac{7571}{16695} & \frac{393}{640} & -\frac{92097}{339200} & \frac{187}{2100} & \frac{1}{40}
\end{array}
$$

- **Verner-Verfahren [Z48]:**

$$
\begin{array}{c|cccccccc}
0 & & & & & & & & \\
\frac{1}{18} & \frac{1}{18} & & & & & & & \\
\frac{1}{6} & -\frac{1}{12} & \frac{1}{4} & & & & & & \\
\frac{2}{9} & -\frac{2}{81} & \frac{4}{27} & \frac{8}{81} & & & & & \\
\frac{2}{3} & \frac{40}{33} & -\frac{4}{11} & -\frac{56}{11} & \frac{54}{11} & & & & \\
1 & -\frac{369}{73} & \frac{72}{73} & \frac{5380}{219} & -\frac{12285}{584} & \frac{2695}{1752} & & & \\
\frac{8}{9} & -\frac{8716}{891} & \frac{656}{297} & \frac{39520}{891} & -\frac{416}{11} & \frac{52}{27} & 0 & & \\
1 & \frac{3015}{256} & -\frac{4}{9} & -\frac{4219}{78} & \frac{5985}{128} & -\frac{539}{384} & 0 & \frac{693}{3328} & \\
\hline
 & \frac{3}{80} & 0 & \frac{4}{25} & \frac{243}{1120} & \frac{77}{160} & \frac{73}{700} & 0 & 0 \\
 & \frac{57}{640} & 0 & -\frac{16}{65} & \frac{1377}{2240} & \frac{121}{320} & 0 & \frac{891}{8320} & \frac{2}{35}
\end{array}
$$

14 Literaturverzeichnis

14.1 Eigene Publikationen

Anmerkung: In unserer Arbeitsgruppe werden die Namen der Autoren in alphabetischer Reihenfolge angegeben.

[E1] Matthias Korch, Thomas Rauber, Matthias Stachowski und Tim Werner.
"Influence of Locality on the Scalability of Method- and System-Parallel Explicit Peer Methods".
In: *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems (FedCSIS).*
2016,
S. 685–694.
DOI: 10.15439/2016F464.

[E2] Matthias Korch und Tim Werner.
"Accelerating Explicit ODE Methods by Kernel Fusion".
In: *Concurrency and Computation: Practice and Experience* 30.18 (2018).
DOI: 10.1002/cpe.4470.

[E3] Matthias Korch und Tim Werner.
"Exploiting Limited Access Distance for Kernel Fusion Across the Stages of Explicit One-Step Methods on GPUs".
In: *30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD).*
2018,
S. 148–157.
DOI: 10.1109/CAHPC.2018.8645892.

[E4] Matthias Korch und Tim Werner.
"Improving Locality of Explicit One-Step Methods on GPUs by Tiling across Stages and Time Steps".
In: *Future Generation Computer Systems* 102 (2020), S. 889–901.
DOI: 10.1016/j.future.2019.07.075.

[E5] Matthias Korch und Tim Werner.
"Multi-Workgroup Tiling to Improve the Locality of Explicit One-Step Methods for ODE Systems with Limited Access Distance on GPUs".
In: *13th International Conference on Parallel Processing and Applied Mathematics (PPAM).*
2019.
DOI: 10.1007/978-3-030-43229-4_1.

[E6] Matthias Korch und Tim Werner.
"An In-Depth Introduction of Multi-Workgroup Tiling for Improving the Locality of Explicit One-Step Methods for ODE Systems with Limited Access Distance on GPUs".
In: *Concurrency and Computation: Practice and Experience* 33.11 (2021), e6016.
DOI: 10.1002/cpe.6016.

[E7] Matthias Korch, Philipp Raithel und Tim Werner.
"Implementation and Optimization of a 1D2V PIC Method for Nonlinear Kinetic Models on GPUs".
In: *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*.
2020,
S. 30–37.
DOI: 10.1109/PDP50117.2020.00012.

14.2 Artikel aus wissenschaftlichen Zeitschriften

[Z1] Syvert Paul Nørsett und Harald H. Simonsen.
"Aspects of Parallel Runge–Kutta methods".
In: 1386 (1989), S. 103–117.
DOI: 10.1007/BFb0089234.

[Z2] Pieter J. van der Houwen und Ben P. Sommeijer.
"Parallel iteration of high-order Runge–Kutta Methods with stepsize control".
In: *Journal of Computational and Applied Mathematics* 29 (1990), S. 111–127.
DOI: 10.1016/0377-0427(90)90200-J.

[Z3] Rüdiger Weiner, Katja Biermann, Stefan Jebens, Bernhard A. Schmitt und Helmut Podhaisky.
"Explicit Two-Step Peer Methods".
In: *Computers & Mathematics with Applications* 55.4 (2008), S. 609–619.
DOI: 10.1016/j.camwa.2007.04.026.

[Z4] Piet J. van der Houwen und Eleonora Messina.
"Parallel Adams Methods".
In: *J. Comput. Appl. Math.* 101.1–2 (1999), S. 153–165.
DOI: 10.1016/S0377-0427(98)00214-3.

[Z5] Yvon Maday und Gabriel Turinici.
"A Parareal in Time Procedure for the Control of Partial Differential Equations".
In: *Comptes Rendus Mathematique* 335.4 (2002), S. 387–392.
DOI: 10.1016/S1631-073X(02)02467-6.

[Z6] C. Lederman, R. Martin und J.-L. Cambier.
"Time-parallel solutions to differential equations via functional optimization".
In: *Computational and Applied Mathematics* 37 (2018), S. 27–51.
DOI: 10.1007/s40314-016-0319-7.

[Z7] Shu-Lin Wu.
"Three Rapidly Convergent Parareal Solvers with Application to Time-Dependent PDEs with Fractional Laplacian".
In: *Mathematical Methods in the Applied Sciences* (2017).
DOI: 10.1002/mma.4273.

[Z8] Christopher A. Kennedy, Mark H. Carpenter und R. Michael Lewis.
"Low-Storage, Explicit Runge–Kutta Schemes for the Compressible Navier–Stokes Equations".
In: *Applied Numerical Mathematics* 35.3 (2000), S. 177–219.
DOI: 10.1016/S0168-9274(99)00141-5.

[Z9] Manuel Calvo, J.M. Franco und Luis Rández.
"A new minimum storage Runge–Kutta scheme for Computational Acoustics".
In: *Journal of Computational Physics* 201.1 (2004), S. 1–12.
DOI: 10.1016/j.jcp.2004.05.012.

[Z10] Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K. Hollingsworth, Boyana Norris und Richard Vuduc.
"Autotuning in High-Performance Computing Applications".

In: *Proceedings of the IEEE* 106.11 (2018), S. 2068–2083.
DOI: 10.1109/JPROC.2018.2841200.

[Z11] Takao Sakurai, Takahiro Katagiri, Hisayasu Kuroda, Ken Naono, Mitsuyoshi Igai und Satoshi Ohshima.
"A Sparse Matrix Library with Automatic Selection of Iterative Solvers and Preconditioners".
In: *Procedia Computer Science* 18 (2013), S. 1332–1341.
DOI: 10.1016/j.procs.2013.05.300.

[Z12] John A. Nelder und Roger Mead.
"A Simplex Method for Function Minimization".
In: *The Computer Journal* 7.4 (1965), S. 308–313.
DOI: 10.1093/comjnl/7.4.308.

[Z13] Martin Pincus.
"Letter to the Editor—A Monte Carlo Method for the Approximate Solution of Certain Types of Constrained Optimization Problems".
In: *Operations Research* 18.6 (1970), S. 1225–1228.
DOI: 10.1287/opre.18.6.1225.

[Z14] Samuel Williams, Andrew Waterman und David Patterson.
"Roofline: An Insightful Visual Performance Model for Multicore Architectures".
In: *Communications of the ACM* 52.4 (2009), S. 65–76.
DOI: 10.1145/1498765.1498785.

[Z15] Jan Fousek, Jiři Filipovič und Matuš Madzin.
"Automatic Fusions of CUDA-GPU Kernels for Parallel Map".
In: *SIGARCH Comput. Archit. News* 39.4 (Dez. 2011), S. 98–99.
DOI: 10.1145/2082156.2082183.

[Z16] Jiří Filipovič, Matúš Madzin, Jan Fousek und Luděk Matyska.
"Optimizing CUDA code by kernel fusion: application on BLAS".
In: *The Journal of Supercomputing* 71.10 (2015), S. 3934–3957.
DOI: 10.1007/s11227-015-1483-z.

[Z17] Karl Rupp, Josef Weinbub, Ansgar Jüngel und Tibor Grasser.
"Pipelined Iterative Solvers with Kernel Fusion for Graphics Processing Units".
In: *ACM Trans. Math. Softw.* 43.2 (2016), 11:1–11:27.
DOI: 10.1145/2907944.

[Z18] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado und Francky Catthoor.
"Polyhedral Parallel Code Generation for CUDA".
In: *ACM Trans. Archit. Code Optim.* 9.4 (2013), 54:1–54:23.
DOI: 10.1145/2400682.2400713.

[Z19] Tareq Malas, Georg Hager, Hatem Ltaief, Holger Stengel, Gerhard Wellein und David Keyes.
"Multicore-Optimized Wavefront Diamond Blocking for Optimizing Stencil Updates".
In: *SIAM Journal on Scientific Computing* 37.4 (2015), S. C439–C464.
DOI: 10.1137/140991133.

[Z20] Vadim Levchenko und Anastasia Perepelkina.
"Locally Recursive Non-Locally Asynchronous Algorithms for Stencil Computation".
In: *Lobachevskii Journal of Mathematics* 39 (Mai 2018), S. 552–561.
DOI: 10.1134/S1995080218040108.

[Z21] Rodrigo C. O. Rocha, Alyson D. Pereira, Luiz Ramos und Luis F. W. Goes.
"TOAST: Automatic Tiling for Iterative Stencil Computations on GPUs".
In: *Concurrency and Computation: Practice and Experience* 29.8 (2017).
DOI: 10.1002/cpe.4053.

[Z22] Thomas Rauber und Gudula Rünger.
"Improving Locality for ODE Solvers by Program Transformations".
In: *Scientific Programming* 12.3 (2004), S. 133–154.
DOI: 10.1155/2004/175169.

[Z23] Thomas Rauber und Gudula Rünger.
"Parallel Implementations of Iterated Runge-Kutta Methods".
In: *The International Journal of Supercomputer Applications and High Performance Computing* 10.1 (1996), S. 62–90.
DOI: 10.1177/109434209601000103.

[Z24] Matthias Korch und Thomas Rauber.
"Optimizing locality and scalability of embedded Runge–Kutta solvers using block-based pipelining".
In: *Journal of Parallel and Distributed Computing* 66.3 (2006), S. 444–468.
DOI: 10.1016/j.jpdc.2005.09.003.

[Z25] Natalia Kalinnik, Matthias Korch und Thomas Rauber.
"Online Auto-Tuning for the Time-Step-Based Parallel Solution of ODEs on Shared-Memory Systems".
In: *Journal of Parallel and Distributed Computing* 74.8 (2014), S. 2722–2744.
DOI: 10.1016/j.jpdc.2014.03.006.

[Z26] Oliver Hacker, Matthias Korch und Johannes Seiferth.
"A Motivating Case Study on Code Variant Selection by Reinforcement Learning".
In: *High Performance Computing*.
2022,
S. 293–312.
DOI: 10.1007/978-3-031-07312-0_15.

[Z27] Arieh Iserles und Syvert Paul Nørsett.
"On the Theory of Parallel Runge—Kutta Methods".
In: *IMA Journal of Numerical Analysis* 10.4 (Okt. 1990), S. 463–488.
DOI: 10.1093/imanum/10.4.463.

[Z28] Norman P. Jouppi.
"Cache Write Policies and Performance".
In: *SIGARCH Comput. Archit. News* 21.2 (1993), S. 191–201.
DOI: 10.1145/173682.165154.

[Z29] Gustaf Söderlind, Kjell Gustafsson und Michael Lundh.
"A PI Stepsize Control for the Numerical Solution of Ordinary Differential Equations".
In: *BIT (Nordisk Tidskrift för Informationsbehandling)* 28.2 (1988), S. 270–287.

DOI: 10.1007/BF01934091.

[Z30] John C. Butcher.
"Coefficients for the study of Runge-Kutta integration processes".
In: *Journal of the Australian Mathematical Society* 3.2 (1963), S. 185–201.
DOI: 10.1017/S1446788700027932.

[Z31] John C. Butcher.
"Implicit Runge-Kutta Processes".
In: *Mathematics of Computation* 18.85 (1964), S. 50–64.
DOI: 10.2307/2003405.

[Z32] Lewis F. Richardson.
"The Approximate Arithmetical Solution by Finite Differences of Physical Problems Involving Differential Equations, with an Application to the Stresses in a Masonry Dam".
In: *Philosophical Transactions of the Royal Society of London Series A* 210 (1911), S. 307–357.
DOI: 10.1098/rsta.1911.0009.

[Z33] Lewis F. Richardson und J. Arthur Gaunt.
"The Deferred Approach to the Limit. Part I. Single Lattice. Part II. Interpenetrating Lattices".
In: *Philosophical Transactions of the Royal Society of London Series A* 226 (1927), S. 299–361.
DOI: 10.1098/rsta.1927.0008.

[Z34] John C. Butcher.
"General linear methods".
In: *Computers & Mathematics with Applications* 31.4 (1996), S. 105–112.
DOI: https://doi.org/10.1016/0898-1221(95)00222-7.

[Z35] R. A. Gingold und Joseph J. Monaghan.
"Smoothed Particle Hydrodynamics: Theory and Application to Non-Spherical Stars".
In: *Monthly Notices of the Royal Astronomical Society* 181.3 (Dez. 1977), S. 375–389.
DOI: 10.1093/mnras/181.3.375.

[Z36] Gerhard Rein und Thomas Rodewis.
"Convergence of a Particle-in-cell Scheme for the Spherically Symmetric Vlasov-Einstein System".
In: *Indiana University Mathematics Journal* 52.4 (2003), S. 821–861.

[Z37] Moncho Gomez-Gesteira, Benedict D. Rogers, Robert A. Dalrymple und Alex J.C. Crespo.
"State-of-the-art of classical SPH for free-surface flows".
In: *Journal of Hydraulic Research* 48 (2010), S. 6–27.
DOI: 10.1080/00221686.2010.9641242.

[Z38] Ryoichi Ando, Nils Thürey und Chris Wojtan.
"Highly Adaptive Liquid Simulations on Tetrahedral Meshes".
In: *ACM Transactions on Graphics* 32.4 (2013).
DOI: 10.1145/2461912.2461982.

[Z39] Daniel Winkler, Massoud Rezavand und Wolfgang Rauch.
"Neighbour lists for smoothed particle hydrodynamics on GPUs".

In: *Computer Physics Communications* 225 (2018), S. 140–148.
DOI: 10.1016/j.cpc.2017.12.014.

[Z40] André R. Brodtkorb, Martin L. Sætra und Mustafa Altinakar.
"Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation".
In: *Computers & Fluids* 55 (2012), S. 1–12.
DOI: 10.1016/j.compfluid.2011.10.012.

[Z41] René Lefever und Grégoire Nicolis.
"Chemical Instabilities and Sustained Oscillations".
In: *Journal of Theoretical Biology* 30.2 (1971), S. 267–284.
DOI: 10.1016/0022-5193(71)90054-3.

[Z42] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins und Peter W. Markstein.
"Register Allocation via Coloring".
In: *Computer Languages* 6.1 (1981), S. 47–57.
DOI: 10.1016/0096-0551(81)90048-5.

[Z43] Christoph Lameter.
"NUMA (Non-Uniform Memory Access): An Overview: NUMA Becomes More Common Because Memory Controllers Get Close to Execution Units on Microprocessors."
In: *Queue* 11.7 (2013), S. 40–51.
DOI: 10.1145/2508834.2513149.

[Z44] Wesley M. Johnston, J. R. Paul Hanna und Richard J. Millar.
"Advances in Dataflow Programming Languages".
In: *ACM Computing Surveys* 36.1 (2004), S. 1–34.
DOI: 10.1145/1013208.1013209.

[Z45] Przemysław Bogacki und Lawrence F. Shampine.
"A 3(2) Pair of Runge-Kutta Formulas".
In: *Applied Mathematics Letters* 2.4 (1989), S. 321–325.
DOI: 10.1016/0893-9659(89)90079-7.

[Z46] Fred H. Chipman.
"A-Stable Runge-Kutta Processes".
In: *BIT Numerical Mathematics* 11 (1971), S. 384–388.
DOI: 10.1007/BF01939406.

[Z47] J.R. Dormand und P.J. Prince.
"A Family of Embedded Runge-Kutta Formulae".
In: *Journal of Computational and Applied Mathematics* 6.1 (1980), S. 19–26.
DOI: 10.1016/0771-050X(80)90013-3.

[Z48] James H. Verner.
"Explicit Runge–Kutta Methods with Estimates of the Local Truncation Error".
In: *SIAM Journal on Numerical Analysis* 15.4 (1978), S. 772–790.
DOI: 10.1137/0715051.

14.3 Beiträge aus wissenschaftlichen Konferenzen

[K1] R. Clint Whaley und Jack J. Dongarra.
"Automatically Tuned Linear Algebra Software".
In: *SC 98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*.
1998,
S. 1–27.
DOI: 10.1109/SC.1998.10004.

[K2] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin und Jim Demmel.
"Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology".
In: *ACM International Conference on Supercomputing 25th Anniversary Volume*.
1997,
S. 253–260.
DOI: 10.1145/2591635.2667174.

[K3] Matteo Frigo und Steven G. Johnson.
"FFTW: An Adaptive Software Architecture for the FFT".
In: *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98*.
Bd. 3.
1998,
S. 1381–1384.
DOI: 10.1109/ICASSP.1998.681704.

[K4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly und Saman Amarasinghe.
"OpenTuner: An Extensible Framework for Program Autotuning".
In: *International Conference on Parallel Architectures and Compilation Techniques*.
2014,
S. 303–315.
DOI: 10.1145/2628071.2628092.

[K5] Cristian Tapus, I-Hsin Chung und Jeffrey K. Hollingsworth.
"Active Harmony: Towards Automated Performance Tuning".
In: *SC 02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*.
2002,
S. 44–44.
DOI: 10.1109/SC.2002.10062.

[K6] Keith Seymour, Haihang You und Jack Dongarra.
"A comparison of search heuristics for empirical code optimization".
In: *2008 IEEE International Conference on Cluster Computing*.
2008,
S. 421–429.
DOI: 10.1109/CLUSTR.2008.4663803.

[K7] Toru Kisuki, Peter M.W. Knijnenburg und Michael F.P. O'Boyle.
"Combined selection of tile sizes and unroll factors using iterative compilation".
In: *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques*.

2000,
S. 237–246.
DOI: 10.1109/PACT.2000.888348.

[K8] Holger Stengel, Jan Treibig, Georg Hager und Gerhard Wellein.
"Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model".
In: *Proceedings of the 29th ACM on International Conference on Supercomputing*.
2015,
S. 207–216.
DOI: 10.1145/2751205.2751240.

[K9] Johannes Hofmann, Dietmar Fey, Michael Riedmann, Jan Eitzinger, Georg Hager und Gerhard Wellein.
"Performance Analysis of the Kahan-Enhanced Scalar Product on Current Multicore Processors".
In: *Parallel Processing and Applied Mathematics*.
2016,
S. 63–73.
DOI: 10.1007/978-3-319-32149-3_7.

[K10] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili und Srimat Chakradhar.
"Optimizing Data Warehousing Applications for GPUs Using Kernel Fusion/Fission".
In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*.
2012,
S. 2433–2442.
DOI: 10.1109/IPDPSW.2012.300.

[K11] Haicheng Wu, Gregory Diamos, Srihari Cadambi und Sudhakar Yalamanchili.
"Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation".
In: *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*.
2012,
S. 107–118.
DOI: 10.1109/MICRO.2012.19.

[K12] Guibin Wang, YiSong Lin und Wei Yi.
"Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU".
In: *IEEE/ACM International Conference on Green Computing and Communications (GreenCom), IEEE/ACM International Conference on Cyber, Physical and Social Computing (CPSCom)*.
2010,
S. 344–350.
DOI: 10.1109/GreenCom-CPSCom.2010.102.

[K13] Mohamed Wahib und Naoya Maruyama.
"Scalable Kernel Fusion for Memory-bound GPU Applications".
In: *SC14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

2014,
S. 191–202.
DOI: 10.1109/SC.2014.21.

[K14] Mohamed Wahib und Naoya Maruyama.
"Automated GPU Kernel Transformations in Large-Scale Production Stencil Applications".
In: *24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.
2015,
S. 259–270.
DOI: 10.1145/2749246.2749255.

[K15] Jiri Filipovic und Siegfried Benkner.
"OpenCL Kernel Fusion for GPU, Xeon Phi and CPU".
In: *27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*.
2015,
S. 98–105.
DOI: 10.1109/SBAC-PAD.2015.29.

[K16] José I. Aliaga, Joaquín Pérez und Enrique S. Quintana-Ortí.
"Systematic Fusion of CUDA Kernels for Iterative Sparse Linear System Solvers".
In: *Euro-Par 2015: Parallel Processing*.
Hrsg. von Jesper Larsson Träff, Sascha Hunold und Francesco Versaci.
2015,
S. 675–686.
DOI: 10.1007/978-3-662-48096-0_52.

[K17] Shucai Xiao, Ashwin M. Aji und Wu-chun Feng.
"On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit".
In: *15th International Conference on Parallel and Distributed Systems (ICPADS)*.
2009,
S. 26–33.
DOI: 10.1109/ICPADS.2009.110.

[K18] Shucai Xiao und Wu-chun Feng.
"Inter-block GPU communication via fast barrier synchronization".
In: *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*.
2010,
S. 1–12.
DOI: 10.1109/IPDPS.2010.5470477.

[K19] Alok Mishra, Martin Kong und Barbara Chapman.
"Kernel Fusion/Decomposition for Automatic GPU-Offloading".
In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.
2019,
S. 283–284.
DOI: 10.1109/CGO.2019.8661188.

[K20] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan und Sven Verdoolaege.

"Split Tiling for GPUs: Automatic Parallelization Using Trapezoidal Tiles".
In: *6th Workshop on General Purpose Processing Using GPUs (GPGPU-6)*.
2013,
S. 24–31.
DOI: 10.1145/2458523.2458526.

[K21] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan und Sven Verdoolaege.
"Hybrid Hexagonal/Classical Tiling for GPUs".
In: *Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.
2014,
S. 66–75.
DOI: 10.1145/2544137.2544160.

[K22] Philip Pfaffe, Tobias Grosser und Martin Tillmann.
"Efficient Hierarchical Online-Autotuning: A Case Study on Polyhedral Accelerator Mapping".
In: *ICS '19: Proceedings of the ACM International Conference on Supercomputing*.
2019,
S. 354–366.
DOI: 10.1145/3330345.3330377.

[K23] Tareq M. Malas, Julian Hornich, Georg Hager, Hatem Ltaief, Christoph Pflaum und David E. Keyes.
"Optimization of an Electromagnetics Code with Multicore Wavefront Diamond Blocking and Multi-dimensional Intra-Tile Parallelization".
In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.
2016,
S. 142–151.
DOI: 10.1109/IPDPS.2016.87.

[K24] Vadim Levchenko, Anastasia Perepelkina, Andrey Zakirov, Ivan Goryachev und Vladimir Savchenko.
"Numerical 3D Simulation of the Light Propagation in the Optical WGM-Microresonator by the FDTD Method".
In: *2017 IEEE International Conference on Computational Electromagnetics (ICCEM)*.
2017,
S. 291–292.
DOI: 10.1109/COMPEM.2017.7912826.

[K25] Boris Korneev und Vadim Levchenko.
"Runge-Kutta Discontinuous Galerkin Method and DiamondTorre GPGPU Algorithm for Effective Simulation of Large 3D Multiphase Fluid Flows with Shocks".
In: *Conference: 2019 International Multi-Conference on Engineering, Computer and Information Sciences (SIBIRCON)*.
2019,
S. 0817–0822.
DOI: 10.1109/SIBIRCON48586.2019.8958102.

[K26] Matthias Christen, Olaf Schenk und Helmar Burkhart.

"PATUS: A Code Generation and Autotuning Framework For Parallel Iterative Stencil Computations on Modern Microarchitectures".
In: *2011 IEEE International Parallel Distributed Processing Symposium*.
2011,
S. 676–687.
DOI: 10.1109/IPDPS.2011.70.

[K27] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk und Charles E. Leiserson.
"The Pochoir Stencil Compiler".
In: *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11)*.
2011,
S. 117–128.
DOI: 10.1145/1989493.1989508.

[K28] Jonathan Ragan-Kelley, Connelly Barnes, Sylvain Paris Andrew Adams, Fredo Durand und Saman Amarasinghe.
"Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines".
In: *Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'13)*.
2013,
S. 519–530.
DOI: 10.1145/2491956.2462176.

[K29] Matthias Korch, Julien Kulbe und Carsten Scholtes.
"Diamond-Like Tiling Schemes for Efficient Explicit Euler on GPUs".
In: *11th International Symposium on Parallel and Distributed Computing (ISPDC)*.
2012,
S. 259–266.
DOI: 10.1109/ISPDC.2012.42.

[K30] Matthias Korch.
"Locality Improvement of Data-Parallel Adams–Bashforth Methods through Block-Based Pipelining of Time Steps".
In: *Euro-Par 2012. Parallel Processing (LNCS 7484)*.
2012,
S. 563–574.
DOI: 10.1007/978-3-642-32820-6_56.

[K31] Johannes Seiferth, Matthias Korch und Thomas Rauber.
"Offsite Autotuning Approach: Performance Model Driven Autotuning Applied to Parallel Explicit ODE Methods".
In: *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings*.
2020,
S. 370–390.
DOI: 10.1007/978-3-030-50743-5_19.

[K32] Dana Petcu.
"Solving Initial Value Problems with a Multiprocessor Code".

In: *Parallel Computing Technologies (PaCT 1999)*.
1999,
S. 452–465.
DOI: 10.1007/3-540-48387-X_47.

[K33] Thomas Rauber, Robert Reilein und Gudula Runger.
"ORT - A Communication Library for Orthogonal Processor Groups".
In: *SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*.
2001,
S. 7–7.
DOI: 10.1109/SC.2001.10015.

[K34] Enzo Rucci, Ezequiel Moreno, Adrián Pousa und Franco Chichizola.
"Optimization of the N-Body Simulation on Intel's Architectures Based on AVX-512 Instruction Set".
In: *Computer Science – CACIC 2019*.
2020,
S. 37–52.
DOI: 10.1007/978-3-030-48325-8_3.

[K35] Vivek Sarkar und Guang R. Gao.
"Optimization of Array Accesses by Collective Loop Transformations".
In: *Proceedings of the 5th International Conference on Supercomputing*.
1991,
S. 194–205.
DOI: 10.1145/109025.109077.

[K36] Jie Chen und William Watson.
"Software Barrier Performance on Dual Quad-Core Opterons".
In: *2008 International Conference on Networking, Architecture, and Storage*.
2008,
S. 303–309.
DOI: 10.1109/NAS.2008.27.

[K37] Liqun Cheng und John B. Carter.
"Fast barriers for scalable ccNUMA systems".
In: *2005 International Conference on Parallel Processing (ICPP'05)*.
2005,
S. 241–250.
DOI: 10.1109/ICPP.2005.39.

[K38] Stijn Eyerman, Wim Heirman und Ibrahim Hur.
"DRAM Bandwidth and Latency Stacks: Visualizing DRAM Bottlenecks".
In: *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.
2022,
S. 322–331.
DOI: 10.1109/ISPASS55109.2022.00045.

14.4 Bücher

[B1] Peter J. M. van Laarhoven und Emile H. L. Aarts.
Simulated Annealing: Theory and Applications.
1. Auflage.
Dordrecht: D. Reidel Publishing Company, 1987.
ISBN: 978-9-027-72513-4.

[B2] David Goldberg.
Genetic Algorithms in Search, Optimization and Machine Learning.
1. Auflage.
Boston, Vereinigte Staaten: Addison-Wesley Professional, 1989.
ISBN: 978-0-201-15767-3.

[B3] John L. Hennessy und David A. Patterson.
Computer Architecture: A Quantitative Approach.
5. Auflage.
Burlington, Massachusetts: Morgan Kaufmann, 2011.
ISBN: 978-8-178-67266-3.

[B4] Walter Oberschelp und Gottfried Vossen.
Rechneraufbau und Rechnerstrukturen.
10. Auflage.
München: Oldenbourg-Verlag, 2006.
ISBN: 978-3-486-57849-2.

[B5] David A. Patterson und John L. Hennessy.
Computer Organization and Design MIPS Edition: The Hardware-Software Interface.
5. Auflage.
Burlington, Massachusetts: Morgan Kaufmann, 2013.
ISBN: 978-0-124-0772-6.

[B6] James Reinders Jim Jeffers und Avinash Sodani.
Intel Xeon Phi Processor High Performance Programming Knights Landing Edition.
2. Auflage.
Burlington, Massachusetts, Vereinigte Staaten: Morgan Kaufmann, 2016.
ISBN: 978-0-128-09194-4.

[B7] Ernst Hairer, Syvert Paul Nørsett und Gerhard Wanner.
Solving Ordinary Differential Equations I: Nonstiff Problems.
2. Auflage.
Berlin: Springer, 2000.
ISBN: 978-3-540-78862-1.

[B8] Kevin Burrage.
Parallel and Sequential Methods for Ordinary Differential Equations.
1. Auflage.
Oxford: Oxford University Press, 1995.
ISBN: 978-0-198-53432-7.

[B9] Karl Strehmel, Rüdiger Weiner und Helmut Podhaisky.
Numerik gewöhnlicher Differentialgleichungen: Nichtsteife, steife und differential-algebraische Gleichungen.

2., überarbeitete und erweiterte Auflage.
Berlin: Springer Spektrum, 2012.
ISBN: 978-3-834-81847-8.

[B10] Francis Bashforth und John Couch Adams.
An attempt to test the theories of capillary action by comparing the theoretical and measured forms of drops of fluid. With an explanation of the method of integration employed in constucting the tables which give the theoretical forms of such drops.
1. Auflage.
Cambridge: Cambridge University Press, 1883.

[B11] Klaus-Jürgen Bathe.
Finite-Elemente-Methoden.
2. Auflage.
London: Springer, 2002.
ISBN: 978-3-540-66806-0.

[B12] William E. Schiesser.
The Numerical Method of Lines: Integration of Partial Differential Equations.
1. Auflage.
London: Academic Press, 1991.
ISBN: 978-0-126-24130-3.

[B13] Farid N. Najm.
Circuit Simulation.
1. Auflage.
Hoboken, New Jersey, Vereinigte Staaten: Wiley, 2010.
ISBN: 978-0-470-53871-5,

[B14] Sverre J. Aarseth.
Gravitational N-Body Simulations.
1. Auflage.
Cambridge: Cambridge University Press, 2003.
ISBN: 978-0-511-53524-6.

[B15] Keith Cooper und Linda Torczon.
Engineering a Compiler.
3. Auflage.
Amsterdam: Elsevier, 2022.
ISBN: 978-0-128-15412-0.

14.5 Technische Spezifikationen, Dokumentationen, Berichte und Whitepaper

[T1] Oleksandr Zinenko, Lorenzo Chelini und Tobias Grosser.
Declarative Transformations in the Polyhedral Model.
Technischer Bericht.
2018.
URL: https://hal.inria.fr/hal-01965599.

[T2] *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210*.
Whitepaper.
NVIDIA. 2014.
URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf.

[T3] *NVIDIA GeForce GTX 980*.
Whitepaper.
NVIDIA. 2014.
URL: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF.

[T4] *NVIDIA Tesla P100*.
Whitepaper.
NVIDIA. 2016.
URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[T5] *NVIDIA Tesla V100 GPU Architecture*.
Whitepaper.
NVIDIA. 2017.
URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[T6] *NVIDIA Turing GPU Architecture*.
Whitepaper.
NVIDIA. 2018.
URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[T7] *NVIDIA A100 Tensor Core GPU Architecture*.
Whitepaper.
NVIDIA. 2020.
URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[T8] *NVIDIA Ampere GA102 GPU Architecture*.
Whitepaper.
NVIDIA. 2021.
URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf.

[T9] *CUDA C++ Programming Guide*.
Dokumentation.
NVIDIA.
URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[T10] *AMD Graphics Cores Next (GCN) Architecture*.
Whitepaper.
AMD. 2012.
URL: https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf.

[T11] *Graphics Core Next Architecture, Generation 3*.
Dokumentation.
AMD. 2016.
URL: http://developer.amd.com/wordpress/media/2013/12/AMD_GCN3_Instruction_Set_Architecture_rev1.1.pdf.

[T12] *RDNA Architecture*.
Whitepaper.
AMD. 2020.
URL: https://www.amd.com/system/files/documents/rdna-whitepaper.pdf.

[T13] *RDNA 1.0 Instruction Set Architecture*.
Dokumentation.
AMD. 2020.
URL: https://developer.amd.com/wp-content/resources/RDNA_Shader_ISA.pdf.

[T14] *AMD CDNA Architecture*.
Whitepaper.
AMD. 2020.
URL: https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf.

[T15] *OpenCL Programming Guide*.
Dokumentation.
AMD.
URL: https://rocmdocs.amd.com/en/latest/Programming_Guides/Opencl-programming-guide.html.

[T16] *Intel OpenSource HD Graphics Programmer's Reference Manual (PRM) Volume 4 Part 3: Execution Unit ISA (Ivy Bridge)*.
Dokumentation.
Intel. 2012.
URL: https://01.org/sites/default/files/documentation/ivb_ihd_os_vol4_part3_0.pdf.

[T17] *The Compute Architecture of Intel Processor Graphics Gen9*.
Whitepaper.
Intel. 2015.

URL: https://www.intel.com/content/dam/develop/external/us/en/documents/the-compute-architecture-of-intel-processor-graphics-gen9-v1d0.pdf.

[T18] *Intel Processor Graphics Gen11 Architecture*.
Whitepaper.
Intel. 2019.
URL: https://www.intel.com/content/dam/develop/external/us/en/documents/the-architecture-of-intel-processor-graphics-gen11-r1new.pdf.

[T19] *Intel oneAPI Programming Guide*.
Dokumentation.
Intel.
URL: https://software.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html.

[T20] Vasily Volkov.
Better Performance at Lower Occupancy.
Technischer Bericht.
UC Berkeley. 2010.
URL: https://www.nvidia.com/content/GTC-2010/pdfs/2238_GTC2010.pdf.

[T21] *C++ AMP: Language and Programming Model*.
Spezifikation.
Microsoft. 2012.
URL: https://download.microsoft.com/download/4/0/E/40EA02D8-23A7-4BD2-AD3A-0BFFFB640F28/CppAMPLanguageAndProgrammingModel.pdf.

[T22] Agner Fog.
The microarchitecture of Intel, AMD, and VIA CPUs.
Technischer Bericht.
2022.
URL: https://www.agner.org/optimize/.

[T23] *Intel 64 and IA-32 Architectures Software Developer's Manual*.
Dokumentation.
Intel. 2022.
URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[T24] *OpenCL™ Developer Guide for Intel® Core™ and Intel® Xeon® Processors*.
Dokumentation.
Intel.
URL: https://rocmdocs.amd.com/en/latest/Programming_Guides/Opencl-programming-guide.html.

[T25] *The OpenCL Specification*.
Spezifikation.
Khronos. 2022.
URL: https://registry.khronos.org/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf.

[T26] *A Guide to Vectorization with Intel® C++ Compilers.*
Dokumentation.
Intel. 2012.
URL: https://www.intel.com/content/dam/develop/external/us/en/documents/compilerautovectorizationguide-635748.pdf.

[T27] *Writing Optimal OpenCL-Code with Intel OpenCL SDK.*
Dokumentation.
Intel. 2011.
URL: https://www.intel.com/content/dam/develop/external/us/en/documents/writing-optimal-opencl-28tm-29-code-with-intel-28r-29-opencl-sdk-184672.pdf.

[T28] Mike O'Connor.
Highlights of the High-Bandwidth Memory (HBM) Standard.
Technischer Bericht.
NVIDIA. 2014.
URL: https://www.cs.utah.edu/thememoryforum/mike.pdf.

14.6 Internetseiten

[W1] Alyson D. Pereira, Rodrigo C. O. Rocha, Luiz Ramos und Luís F. W. Góes. *PSkel: A high-performance framework for stencil computations*. URL: https://pskel.github.io/.

[W2] Intel, Hrsg. *YASK: Yet Another Stencil Kit*. URL: https://github.com/intel/yask.

[W3] ExaStencils Consortium, Hrsg. *Advanced Stencil-Code Engineering (ExaStencils)*. URL: http://www.exastencils.org/.

[W4] Karsten Ahnert und Mario Mulansky. *odeint – solving ODEs in C++*. URL: http://headmyshoulder.github.io/odeint-v2/.

[W5] *Boost C++ Libraries*. URL: http://www.boost.org/.

[W6] NVIDIA, Hrsg. *Thrust*. URL: https://developer.nvidia.com/thrust.

[W7] Matthias Korch. *Auszug aus der E2L-Bibliothek*. URL: https://github.com/UBT-AI2/rk.

[W8] Anton Shilov. *AMD Bulldozer 'Core' Lawsuit: AMD Settles for $12.1m, Payouts for Some*. URL: https://www.anandtech.com/show/14804/amd-settlement.

[W9] NVIDIA, Hrsg. *CUDA*. URL: https://developer.nvidia.com/cuda-zone.

[W10] Khronos Group, Hrsg. *OpenCL*. URL: https://www.khronos.org/opencl/.

[W11] Microsoft, Hrsg. *Direct X Dokumentation*. URL: https://learn.microsoft.com/de-de/windows/win32/directx.

[W12] Khronos Group, Hrsg. *Vulkan*. URL: https://www.vulkan.org/.

[W13] Khronos Group, Hrsg. *OpenGL*. URL: https://www.opengl.org/.

[W14] AMD, Hrsg. *HIP: C++ Heterogeneous-Compute Interface for Portability*. URL: https://github.com/ROCm-Developer-Tools/HIP.

[W15] *WikiChip*.
URL: https://en.wikichip.org.

[W16] *CPP-reference*.
URL: https://en.cppreference.com/.

[W17] Intel, Hrsg.
Intel oneAPI DPC++/C++ Compiler.
URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html.

[W18] Microsoft, Hrsg.
Microsoft C++, C, and Assembler documentation.
URL: https://learn.microsoft.com/en-us/cpp/?view=msvc-170.

[W19] GCC steering committee, Hrsg.
GCC, the GNU Compiler Collection.
URL: https://gcc.gnu.org/.

[W20] NVIDIA, Hrsg.
NVRTC.
URL: https://docs.nvidia.com/cuda/nvrtc/.

[W21] Erez Shinan.
Lark - a parsing toolkit for Python.
URL: https://github.com/lark-parser/lark.

[W22] *DOT Language*.
URL: https://www.graphviz.org/doc/info/lang.html.

[W23] *GitHub*.
URL: https://github.com/.

15 Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die von mir angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin erkläre ich, dass ich die Hilfe von gewerblichen Promotionsberatern beziehungsweise Promotionsvermittlern oder ähnlichen Dienstleistern weder bisher in Anspruch genommen habe, noch künftig in Anspruch nehmen werde.

Zusätzlich erkläre ich hiermit, dass ich keinerlei frühere Promotionsversuche unternommen habe.

Bayreuth, den 30. Mai 2023

____

Unterschrift